Copyright and Trademarks

Version 7.1
September 2017

Copyright © 2017 by LispWorks Ltd.

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of LispWorks Ltd.

The information in this publication is provided for information only, is subject to change without notice, and should not be construed as a commitment by LispWorks Ltd. LispWorks Ltd assumes no responsibility or liability for any errors or inaccuracies that may appear in this publication. The software described in this book is furnished under license and may only be used or copied in accordance with the terms of that license.

LispWorks and KnowledgeWorks are registered trademarks of LispWorks Ltd.

Adobe and PostScript are registered trademarks of Adobe Systems Incorporated. Other brand or product names are the registered trademarks or trademarks of their respective holders.


The XP Pretty Printer bears the following copyright notice, which applies to the parts of LispWorks derived therefrom:

Copyright © 1989 by the Massachusetts Institute of Technology, Cambridge, Massachusetts.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that this copyright and permission notice appear in all copies and supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representation about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty. M.I.T. disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall M.I.T. be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

LispWorks contains part of ICU software obtained from http://source.icu-project.org and which bears the following copyright and permission notice:

ICU License - ICU 1.8.1 and later
COPYRIGHT AND PERMISSION NOTICE
Copyright © 1995-2006 International Business Machines Corporation and others. All rights reserved.
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.
THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder. All trademarks and registered trademarks mentioned herein are the property of their respective owners.

US Government Restricted Rights
The LispWorks Software is a commercial computer software program developed at private expense and is provided with restricted rights. The LispWorks Software may not be used, reproduced, or disclosed by the Government except as set forth in the accompanying End User License Agreement and as provided in DFARS 227.7202-1(a), 227.7202-3(a) (1995), FAR 12.212(a)(1995), FAR 52.227-19, and/or FAR 52.227-14 Alt III, as applicable. Rights reserved under the copyright laws of the United States.

Address
LispWorks Ltd
St. John’s Innovation Centre
Cowley Road
Cambridge
CB4 0WS
England

Telephone
From North America: 877 759 8839
(toll-free)
From elsewhere: +44 1223 421860

Fax
From North America: 617 812 8283
From elsewhere: +44 870 2206189

www.lispworks.com
Contents

1 Starting LispWorks  1
   The usual way to start LispWorks  1
   Passing arguments to LispWorks  1
   Starting the LispWorks Graphical IDE  3
   Using LispWorks with SLIME  3
   Quitting LispWorks  4

2 The Listener  5
   First use of the listener  5
   Standard listener commands  6
   The listener prompt  8

3 The Debugger  9
   Entering the REPL debugger  10
   Simple use of the REPL debugger  11
   The stack in the debugger  12
   REPL debugger commands  13
   Debugger troubleshooting  23
   Debugger control variables  23
   Remote debugging  25

4 The REPL Inspector  37
Contents

Describe 37
Inspect 38
Inspection modes 40

5 The Trace Facility 47
Simple tracing 47
Tracing options 49
Example 56
Tracing methods 57
Tracing subfunctions 58
Trace variables 59
Troubleshooting tracing 60

6 The Advice Facility 63
Defining advice 64
Combining the advice 65
Removing advice 66
Advice for macros and methods 67
Advising subfunctions 69
Examples 70
Advice functions and macros 73

7 Dspecs: Tools for Handling Definitions 75
Dspecs 75
Forms of dspecs 76
Dspec namespaces 77
Types of relations between definitions 81
Details of built-in dspec classes and aliases 82
Subfunction dspecs 85
Tracking definitions 87
Finding locations 89
Users of location information 89

8 Action Lists 95
Defining action lists and actions 96
Exception handling variables 96
Other variables 97
Diagnostic utilities 97
Examples 97
Standard Action Lists 99

9 The Compiler 101
Compiling a function 102
Compiling a source file 103
Compiling a form 103
How the compiler works 103
Compiler control 104
Declare, proclaim, and declaim 109
Optimizing your code 110
Compiler parameters affecting LispWorks 116

10 Code Coverage 119
Using Code Coverage 119
Manipulating code coverage data 121
Preventing code generation for some forms 122
Code coverage and multithreading 123
Counting overflow 123
Memory usage and code speed 123
Understanding the code coverage output 124
Coloring code that has changed 127

11 Memory Management 129
Introduction 129
Guidance for control of the memory management system 130
Memory Management in 32-bit LispWorks 135
Memory Management in 64-bit LispWorks 146
The Mobile GC 151
Common Memory Management Features 163

12 The Profiler 167
What the profiler does 167
Setting up the profiler 168
Running the profiler 169
Profiler output 170
Profiling pitfalls 172
Profiling and garbage collection 173
Profiler tree file format 174

13 Customization of LispWorks 177
Introduction 177
Configuration and initialization files 178
Saving a LispWorks image 179
Saved sessions 182
Load and open your files on startup 184
Customizing the editor 185
Finding source code 186
Controlling redefinition warnings 187
Specifying the initial working directory 187
Using ! for :redo 188
Customizing LispWorks for use with your own code 188
Structure printing 189
Configuring the printer 189

14 LispWorks as a dynamic library 191
Introduction 191
Creating a dynamic library 192
Initialization of the dynamic library 193
Relocation 194
Multiprocessing in a dynamic library 195
Unloading a dynamic library 195

15 Java interface 197
Types and conversion between Lisp and Java 198
Calling from Lisp to Java 200
Calling from Java to Lisp 206
Working with Java arrays 210
Initialization of the Java interface 213
Utilities and administration 213
Loading a LispWorks dynamic library into Java 217
CLOS partial integration 218
Java interface performance issues 220
16 Android interface 223
   Delivering for Android 224
   Directories on Android 227
   The Othello demo for Android 227

17 iOS interface 241
   Delivering for iOS 241
   Initializing LispWorks 243
   Using Objective-C from Lisp 243
   Limitations of the iOS Runtime 243
   The Othello demo for iOS 244
   The Mobile GC 247

18 The Metaobject Protocol 249
   Metaobject features incompatible with AMOP 249
   Metaobject features additional to AMOP 252
   Common problems when using the MOP 252
   Implementation of virtual slots 253

19 Multiprocessing 261
   Introduction to processes 261
   Processes basics 262
   Atomicity and thread-safety of the LispWorks implementation 265
   Locks 277
   Modifying a hash table with multiprocessing 279
   Process Waiting and communication between processes 280
   Synchronization between threads 285
   Killing a process, interrupts and blocking interrupts 288
   Timers 293
   Process properties 294
   Other processes functions 294
   Native threads and foreign code 296
   Low level operations 299

20 Common Defsystem and ASDF 305
   Introduction 305
   Defining a system 306
## Contents

Using ASDF 311

### 21 The Parser Generator 313
- Introduction 313
- Grammar rules 313
- Functions defined by defparser 315
- Error handling 316
- Interface to lexical analyzer 317
- Example 317

### 22 Dynamic Data Exchange 321
- Introduction 321
- Client interface 323
- Server interface 327

### 23 Common SQL 331
- Introduction 331
- Initialization 334
- Functional interface 346
- Object oriented interface 354
- Symbolic SQL syntax 358
- Working with date fields 370
- SQL I/O recording 372
- Error handling in Common SQL 372
- Using MySQL 373
- Using Oracle 379
- Oracle LOB interface 380
- Using ODBC 390
- Using SQLite 392

### 24 User Defined Streams 399
- Introduction 399
- An illustrative example of user defined streams 399

### 25 TCP and UDP socket communication and SSL 407
- Running a server that accepts connections 407
- Connecting to a server 408
Specifying the target for connecting and binding a socket 408
Information about IP addresses 409
Waiting on a socket stream 410
Special considerations 410
Asynchronous I/O 411
Using SSL 418
Socket streams with Java sockets and SSL on Android 432

26 Internationalization: characters, strings and encodings 435

Introduction 435
Unicode support 435
Character and String types 436
String accessors 440
String Construction 441
External Formats 443
16-bit External formats guide 445
External Formats and File Streams 447
External Formats and the Foreign Language Interface 450
Unicode character and string functions 451

27 LispWorks’ Operating Environment 453

The Operating System 453
Site Name 453
The Lisp Image 454
The Command Line 454
Address Space and Image Size 458
Startup relocation 459
Calling external programs 462
Snapshot debugging of startup errors 463
System message log 463
Exit status 463
Creating a new executable with code preloaded 464
User Preferences 464
File system interface 465
Special locations in the file system 466
Accessing the Windows registry 468
Contents

Pathname comparison on Mac OS X 469

28 Miscellaneous Utilities 471
  Object addresses and memory 471
  Optimized integer arithmetic and integer vector access 472
  Transferring large amounts of data 476
  Rings 476
  Conditional throw and checking for catch in the dynamic environment 476
  Checking for a dynamic binding 476

29 64-bit LispWorks 477
  Introduction 477
  Heap size 478
  Architectural constants 479
  Speed 479
  Memory Management and cl:room 479
  Greater allocation expected in 64-bit LispWorks 479
  Float types 480
  External libraries 480

30 Self-contained examples 481
  COMM examples 481
  Streams examples 482
  DDE examples 482
  Parser generator examples 482
  Examples for save-image in a Mac OS X application bundle 482
  Miscellaneous examples 483

31 The CLOS Package 485
  break-new-instances-on-access 485
  break-on-access 486
  class-extra-initargs 487
  compute-class-potential-initargs 488
  compute-discriminating-function 490
  compute-effective-method-function-from-classes 490
  copy-standard-object 493
  funcallable-standard-object 494
Contents

process-a-class-option 495
process-a-slot-option 497
replace-standard-object 499
set-clos-initarg-checking 501
set-make-instance-argument-checking 502
slot-boundp-using-class 502
slot-makunbound-using-class 503
slot-value-using-class 504
trace-new-instances-on-access 505
trace-on-access 506
unbreak-new-instances-on-access 509
unbreak-on-access 509
untrace-new-instances-on-access 510
untrace-on-access 510

32 The COMM Package 513
accept-tcp-connections-creating-async-io-states 513
accepting-handle 517
accepting-handle-collection 518
accepting-handle-name 519
accepting-handle-local-port 520
accepting-handle-socket 520
accepting-handle-user-info 521
apply-in-wait-state-collection-process 522
async-io-state 523
async-io-state-abort 523
async-io-state-abort-and-close 525
async-io-state-address 526
async-io-state-attach-ssl 526
async-io-state-buffered-data-length 528
async-io-state-ctx 529
async-io-state-detach-ssl 529
async-io-state-collection 530
async-io-state-discard 531
async-io-state-finish 532
async-io-state-get-buffered-data 533
async-io-state-handshake 534
async-io-state-max-read 535
async-io-state-name 536
Contents

async-io-state-object 536
async-io-state-old-length 537
async-io-state-peer-address 538
async-io-state-read-buffer 538
async-io-state-read-status 540
async-io-state-write-status 540
async-io-state-read-timeout 541
async-io-state-write-timeout 541
async-io-state-read-with-checking 542
async-io-state-receive-message 545
async-io-state-send-message 547
async-io-state-send-message-to-address 549
async-io-state-ssl 551
async-io-state-ssl-side 551
async-io-state-write-buffer 552
async-io-state-user-info 553
attach-ssl 554
call-wait-state-collection 556
close-accepting-handle 557
close-async-io-state 558
close-wait-state-collection 559
create-and-run-wait-state-collection 560
create-async-io-state 562
create-async-io-state-and-connected-tcp-socket 564
create-async-io-state-and-connected-udp-socket 567
create-async-io-state-and-udp-socket 569
connect-to-tcp-server 572
destroy-ssl 574
destroy-ssl-ctx 575
detach-ssl 575
do-rand-seed 576
ensure-ssl 577
get-default-local-ipv6-address 578
get-host-entry 578
get-ip-default-zone-id 581
get-service-entry 582
get-socket-address 583
get-socket-peer-address 584
get-verification-mode 585
ip-address-string 585
ipv6-address 586
ipv6-address-p 587
ipv6-address-scope-id 587
ipv6-address-string 588
make-ssl-ctx 588
loop-processing-wait-state-collection 589
make-wait-state-collection 590
open-tcp-stream 591
open-tcp-stream-using-java 595
openssl-version 599
parse-ipv6-address 600
pem-read 602
read-dhparams 603
replace-socket-stream-socket 604
server-terminate 605
set-verification-mode 606
set-ssl-ctx-dh 608
set-ssl-ctx-options 610
set-ssl-ctx-password-callback 610
set-ssl-library-path 611
socket-error 612
socket-stream 612
socket-stream-address 617
socket-stream-ctx 618
socket-stream-handshake 618
socket-stream-peer-address 619
socket-stream-shutdown 620
socket-stream-ssl 621
socket-stream-ssl-side 622
ssl-cipher-pointer 622
ssl-cipher-pointer-stack 622
ssl-closed 623
ssl-condition 623
ssl-ctx-pointer 624
ssl-error 624
ssl-failure 624
ssl-new 625
ssl-pointer 625
ssl-x509-lookup 626
start-up-server 626
start-up-server-and-mp 632
string-ip-address 633
switch-open-tcp-stream-with-ssl-to-java 634
wait-for-wait-state-collection 635
wait-state-collection 636
wait-state-collection-stop-loop 636
with-noticed-socket-stream 637

33 The COMMON-LISP Package 639

apropos 639
apropos-list 640
base-string 641
simple-base-string 641
close 641
coerce 642
compile 643
compile-file 645
concatenate 650
declaim 650
declare 651
defclass 657
defpackage 661
describe 663
directory 663
disassemble 668
documentation 668
double-float 669
*features* 670
input-stream-p 676
interactive-stream-p 676
load-logical-pathname-translations 677
long-float 678
long-site-name 678
loop 679
make-array 680
make-hash-table 682
make-instance 684
make-sequence 685
make-string 686
make-string-output-stream 687
map 688
merge 688
open 689
open-stream-p 691
output-stream-p 692
proclaim 693
restart-case 694
room 695
short-float 700
short-site-name 701
simple-string 702
single-float 703
software-type 703
software-version 704
*standard-input* 705
*standard-output* 705
*trace-output* 705
*error-output* 705
*query-io* 705
*debug-io* 705
step 706
stream-element-type 709
string 710
time 711
trace 713
truename 720
untrace 721
update-instance-for-different-class 722
update-instance-for-redefined-class 723
with-output-to-string 723

34 The COMPILER Package 725
deftransform 725

35 The DBG Package 729
close-remote-debugging-connection 729
configure-remote-debugging-spec 730
create-client-remote-debugging-connection 734
create-ide-remote-debugging-connection 734
*debug-print-length* 738
*debug-print-level* 738
*default-client-remote-debugging-server-port* 739
*default-ide-remote-debugging-server-port* 740
ensure-remote-debugging-connection 741
executable-log-file 741
*hidden-packages* 742
ide-attach-remote-output-stream 744
ide-connect-remote-debugging 745
ide-eval-form-in-remote 747
ide-funcall-in-remote 747
ide-find-remote-debugging-connection 750
ide-set-default-remote-debugging-connection 750
ide-list-remote-debugging-connections 750
ide-open-a-listener 753
log-bug-form 754
logs-directory 755
output-backtrace 756
*print-binding-frames* 757
*print-catch-frames* 759
*print-handler-frames* 760
*print-invisible-frames* 762
*print-open-frames* 763
*print-restart-frames* 764
remote-debugging-connection 765
client-remote-debugging 765
ide-remote-debugging 765
remote-debugging-connection-add-close-cleanup 765
remote-debugging-connection-remove-close-cleanup 765
remote-debugging-connection-name 767
remote-debugging-connection-peer-address 768
remote-debugging-stream-peer-address 769
remote-inspect 770
remote-object-p 772
remote-object-connection 772
set-debugger-options 773
set-default-remote-debugging-connection 774
set-remote-debugging-connection 775
Contents

start-client-remote-debugging-server 776
start-ide-remote-debugging-server 779
start-remote-listener 781
*terminal-debugger-block-multiprocessing* 783
with-debugger-wrapper 785
with-remote-debugging-connection 788
with-remote-debugging-spec 789

36 The DSPEC Package 791
*active-finders* 791
at-location 792
canonicalize-dspec 793
def 794
define-dspec-alias 795
define-dspec-class 796
define-form-parser 799
discard-source-info 803
dspec-class 803
*dspec-classes* 804
dspec-defined-p 804
dspec-definition-locations 805
dspec-equal 806
dspec-name 806
dspec-primary-name 807
dspec-progenitor 808
dspec-subclass-p 808
dspec-undefineder 809
find-dspec-locations 809
find-name-locations 810
get-form-parser 811
local-dspec-p 812
location 813
name-defined-dspecs 813
name-definition-locations 814
name-only-form-parser 815
object-dspec 816
parse-form-dspec 817
record-definition 817
*record-source-files* 818
Contents

*redefinition-action* 819
replacement-source-form 820
save-tags-database 821
single-form-form-parser 822
single-form-with-options-form-parser 822
traceable-dspec-p 823
tracing-enabled-p 824
tracing-state 825

37 The EXTERNAL-FORMAT Package 827
  :bmp 827
  :bmp-native 827
  :bmp-reversed 827
  :unicode 829
  :utf-16 830
  :utf-16be 830
  :utf-16le 830
  :utf-16-native 830
  :utf-16-reversed 830
  :utf-32 831
  :utf-32le 831
  :utf-32be 831
  :utf-32-native 831
  :utf-32-reversed 831
char-external-code 833
decode-external-string 834
encode-lisp-string 835
external-format-error 835
external-format-foreign-type 836
external-format-type 836
find-external-char 837
valid-external-format-p 838

38 The HCL Package 839
add-code-coverage-data 839
reverse-subtract-code-coverage-data 839
subtract-code-coverage-data 839
destructive-add-code-coverage-data 839
destructive-subtract-code-coverage-data 839
destructive-reverse-subtract-code-coverage-data 839
add-special-free-action 841
add-symbol-profiler 842
allocation-in-gen-num 843
analysing-special-variables-usage 844
android-build-value 848
android-funcall-in-main-thread 849
android-funcall-in-main-thread-list 849
android-get-current-activity 851
*android-main-process-for-testing* 852
android-main-thread-p 852
any-capi-window-displayed-p 853
array-single-thread-p 854
array-weak-p 854
augment-environment 855
avoid-gc 857
*background-input* 858
*background-output* 858
*background-query-io* 858
binds-who 860
block-promotion 861
building-universal-intermediate-p 862
calls-who 863
cd 864
change-directory 864
check-fragmentation 865
clean-down 866
clean-generation-0 868
clear-code-coverage 869
reset-code-coverage 869
restore-code-coverage-data 869
code-coverage-data 870
code-coverage-data-generate-coloring-html 871
code-coverage-data-generate-statistics 877
code-coverage-file-stats 879
code-coverage-set-editor-colors 881
code-coverage-set-editor-default-data 883
code-coverage-set-html-background-colors 884
collect-generation-2 885
collect-highest-generation 886
Contents

*compiler-break-on-error* 887
compile-file-if-needed 887
copy-code-coverage-data 889
copy-current-code-coverage 889
load-code-coverage-data 889
save-code-coverage-data 889
save-current-code-coverage 889
copy-to-weak-simple-vector 891
create-macos-application-bundle 892
create-temp-file 895
open-temp-file 895
create-universal-binary 897
current-function-name 897
current-stack-length 898
date-string 899
declaration-information 900
*default-package-use-list* 901
*default-profiler-collapse* 901
*default-profiler-cutoff* 902
*default-profiler-limit* 902
*default-profiler-sort* 903
defglobal-parameter 903
defglobal-variable 904
define-declaration 905
delete-advice 907
delivered-image-p 908
deliver-to-android-project 909
*disable-trace* 914
do-profiling 915
editor-color-code-coverage 917
enlarge-generation 920
enlarge-static 921
ensure-hash-entry 922
error-situation-forms 923
expand-generation-1 924
extend-current-stack 925
extended-time 926
fasl-error 929
fast-directory-files 930
fdf-handle-directory-p 930
fdf-handle-directory-string 930
fdf-handle-last-access 930
fdf-handle-last-modify 930
fdf-handle-link-p 930
fdf-handle-size 930
fdf-handle-writable-p 930
file-string 932
file-writable-p 933
filter-code-coverage-data 933
find-object-size 935
find-throw-tag 936
finish-heavy-allocation 936
flag-not-special-free-action 937
flag-special-free-action 938
function-information 939
get-code-coverage-delta 940
reset-code-coverage-snapshot 940
set-code-coverage-snapshot 940
gc-generation 942
gc-if-needed 946
generate-code-coverage 946
get-default-generation 949
get-gc-parameters 950
get-temp-directory 951
get-working-directory 952
gethash-ensuring 952
*handle-existing-defpackage* 954
*handle-old-in-package* 955
*handle-old-in-package-used-as-make-package* 956
hash-table-weak-kind 957
load-data-file 957
*load-fasl-or-lisp-file* 961
make-ring 962
make-unlocked-queue 964
unlocked-queue-read 964
unlocked-queue-peek 964
unlocked-queue-ready 964
unlocked-queue-send 964
unlocked-queue-count 964
unlocked-queue-size 964
Contents

map-code-coverage-data 966
map-ring 967
mark-and-sweep 967
*max-trace-indent* 969
merge-code-coverage-data 970
destructive-merge-code-coverage-data 970
modify-hash 971
normal-gc 972
*packages-for-warn-on-redefinition* 973
parse-float 975
position-in-ring 976
position-in-ring-forward 976
print-profile-list 977
*print-string* 981
profile 981
*profiler-threshold* 982
*profile-symbol-list* 983
profiler-tree-from-function 983
profiler-tree-to-function 984
reduce-memory 985
references-who 987
remove-special-free-action 988
remove-symbol-profiler 989
reset-profiler 989
reset-ring 990
ringp 991
ring-length 991
ring-name 992
ring-pop 992
ring-push 993
ring-ref 994
rotate-ring 995
safe-format-to-string 996
safe-format-to-limited-string 996
safe-prin1-to-string 996
safe-princ-to-string 996
save-argument-real-p 997
save-current-profiler-tree 998
save-current-session 1000
save-image 1001
Contents

save-image-with-bundle 1010
save-universal-from-script 1011
set-array-single-thread-p 1012
set-array-weak 1012
set-default-generation 1014
set-gc-parameters 1015
set-hash-table-weak 1017
set-minimum-free-space 1019
set-process-profiling 1020
set-profiler-threshold 1022
set-promotion-count 1023
set-system-message-log 1024
set-up-profiler 1025
sets-who 1030
source-debugging-on-p 1031
start-gc-timing 1031
stop-gc-timing 1031
get-gc-timing 1031
start-profiling 1033
string-trim-whitespace 1035
stop-profiling 1036
sweep-all-objects 1037
switch-static-allocation 1038
*symbol-alloc-gen-num* 1039
symbol-dynamically-bound-p 1039
throw-if-tag-found 1040
toggle-source-debugging 1040
total-allocation 1042
*traced-arglist* 1043
*traced-results* 1043
*trace-indent-width* 1044
*trace-level* 1045
*trace-print-circle* 1046
*trace-print-length* 1047
*trace-print-level* 1048
*trace-print-pretty* 1049
*trace-verbose* 1050
try-compact-in-generation 1051
try-move-in-generation 1052
undefine-declaration 1054
Contents

unlocked-queue 1055
unwind-protect-blocking-interrupts 1055
unwind-protect-blocking-interrupts-in-cleanups 1057
variable-information 1058
who-binds 1059
who-calls 1060
who-references 1061
who-sets 1061
with-code-coverage-generation 1062
with-ensuring-gethash 1063
with-hash-table-locked 1065
with-heavy-allocation 1066
with-output-to-fasl-file 1067
dump-form 1067
dump-forms-to-file 1067
with-ring-locked 1071
without-code-coverage 1071
write-string-with-properties 1072

39 The LINK-LOAD Package 1075
break-on-unresolved-functions 1075
foreign-symbol-address 1076
get-foreign-symbol 1077
lisp-name-to-foreign-name 1078
read-foreign-modules 1078

40 The LISPWORKS Package 1081
8-bit-string 1081
16-bit-string 1082
append-file 1082
appendf 1083
*autoload-asdf-integration* 1083
base-character 1084
base-character-p 1084
base-char-p 1085
base-char-code-limit 1085
base-string-p 1086
simple-base-string-p 1086
bmp-char 1086
Contents

bmp-char-p 1087
bmp-string 1088
simple-bmp-string 1088
bmp-string-p 1089
simple-bmp-string-p 1089
*browser-location* 1089
call-next-advice 1090
choose-unicode-string-hash-function 1092
compile-system 1092
concatenate-system 1094
copy-file 1096
count-regexp-occurrences 1097
current-pathname 1099
defadvice 1100
*default-action-list-sort-time* 1103
*default-character-element-type* 1104
define-action 1105
define-action-list 1107
defsystem 1109
*defs system-verbose* 1115
delete-directory 1115
deliver 1116
*describe-length* 1117
*describe-level* 1117
*describe-print-length* 1119
*describe-print-level* 1119
dll-quit 1120
dotted-list-length 1122
dotted-list-p 1123
do-nothing 1123
*enter-debugger-directly* 1124
environment-variable 1125
ererrno-value 1126
example-file 1127
example-compile-file 1127
example-edit-file 1128
example-load-binary-file 1129
execute-actions 1129
extended-char 1131
extended-character 1131
Contents

extended-character-p 1132
extended-char-p 1132
*external-formats* 1133
false 1134
file-directory-p 1134
find-regexp-in-string 1135
function-lambda-list 1137
get-inspector-values 1138
sort-inspector-p 1138
get-unix-error 1142
*grep-command* 1142
*grep-command-format* 1143
*grep-fixed-args* 1144
*handle-existing-action-in-action-list* 1144
*handle-existing-action-list* 1145
*handle-missing-action-list* 1145
*handle-missing-action-in-action-list* 1146
*handle-warn-on-redefinition* 1146
hardcopy-system 1147
*init-file-name* 1148
*inspect-through-gui* 1149
lisp-image-name 1149
*lispworks-directory* 1150
load-all-patches 1152
load-system 1153
make-unregistered-action-list 1154
make-mt-random-state 1156
mt-random 1157
*mt-random-state* 1157
mt-random-state 1158
mt-random-state-p 1158
pathname-location 1159
precompile-regexp 1160
print-actions 1161
print-action-lists 1161
*print-command* 1162
*print-nickname* 1162
*prompt* 1163
push-end 1164
push-end-new 1164
quit 1165
rebinding 1167
regexp-find-symbols 1168
remove-advice 1169
removef 1171
*require-verbose* 1171
rotate-byte 1172
round-to-single-precision 1172
sbchar 1173
sequencep 1174
set-default-character-element-type 1174
set-quit-when-no-windows 1176
simple-char 1176
simple-char-p 1177
split-sequence 1177
split-sequence-if 1178
split-sequence-if-not 1179
start-tty-listener 1180
stchar 1181
string-append 1182
string-append* 1183
structurep 1184
text-string 1185
simple-text-string 1185
text-string-p 1186
simple-text-string-p 1186
ttrue 1186
undefine-action 1187
undefine-action-list 1188
unicode-alpha-char-p 1188
unicode-alphanumeric-p 1189
unicode-both-case-p 1190
unicode-char-equal 1191
unicode-char-not-equal 1191
unicode-char-greaterp 1192
unicode-char-lessp 1192
unicode-char-not-greaterp 1193
unicode-char-not-lessp 1193
unicode-lower-case-p 1194
unicode-string-equal 1194
Contents

unicode-string-not-equal 1194
unicode-string-greaterp 1196
unicode-string-lessp 1196
unicode-string-not-greaterp 1197
unicode-string-not-lessp 1197
unicode-upper-case-p 1198
user-preference 1199
when-let 1201
when-let* 1202
whitespace-char-p 1203
with-action-item-error-handling 1204
with-action-list-mapping 1205
with-unique-names 1206

41 The LW-JI Package 1209

call-java-method 1209
call-java-method-error 1210
catching-java-exceptions 1211
catching-exceptions-bind 1211
check-lisp-calls-initialized 1213
create-instance-from-jobject 1213
create-instance-jobject-list 1214
create-instance-jobject 1214
create-java-object 1216
create-java-object-error 1217
default-constructor-arguments 1217
default-name-constructor 1218
define-field-accessor 1219
define-java-caller 1220
define-java-constructor 1220
define-java-callers 1223
define-lisp-proxy 1225
ensure-lisp-classes-from-tree 1233
ensure-supers-contain-java.lang.object 1235
field-access-exception 1236
field-exception 1237
find-java-class 1238
format-to-java-host 1239
generate-java-class-definitions 1239
get-host-java-virtual-machine 1245
get-java-virtual-machine 1246
get-jobject 1247
ensure-is-jobject 1247
get-primitive-array-region 1248
set-primitive-array-region 1248
get-superclass-and-interfaces-tree 1250
import-java-class-definitions 1251
init-java-interface 1253
setup-java-interface-callbacks 1253
intern-and-export-list 1260
jaref 1260
java-array-element-type 1262
java-array-error 1263
java-array-indices-error 1263
java-array-length 1264
java-array-simple-error 1265
java-bad-jobject 1265
java-definition-error 1266
java-class-error 1266
java-method-error 1266
java-field-error 1266
java-exception 1267
java-field-setting-error 1268
java-id-exception 1269
java-instance-without-jobject-error 1270
java-interface-error 1270
java-low-level-exception 1271
java-method-exception 1271
java-normal-exception 1272
java-not-a-java-object-error 1273
java-not-an-array-error 1274
java-null* 1274
java-object-array-element-type 1275
java-objects-eq 1275
java-out-of-bounds-error 1276
java-storing-wrong-type-error 1276
java-primitive-array-element-type 1277
java-serious-exception 1278
java-type-to-lisp-array-type 1278
Contents

lisp-array-type-to-java-type 1278
java-vm-poi 1280
jboolean 1280
jbyte 1280
jchar 1280
jdouble 1280
jfloat 1280
jint 1280
jlong 1280
jshort 1280
jni-env-poi 1281
jobject 1281
jobject-class-name 1282
jobject-ensure-global 1283
jobject-of-class-p 1284
jobject-p 1285
jobject-pretty-class-name 1285
jobject-string 1286
jobject-to-lisp 1287
jvref 1288
lisp-java-instance-p 1290
lisp-to-jobject 1290
make-java-array 1291
make-java-instance 1292
make-lisp-proxy 1293
make-lisp-proxy-with-overrides 1293
map-java-object-array 1295
primitive-array-to-lisp-array 1299
lisp-array-to-primitive-array 1299
read-java-field 1302
checked-read-java-field 1302
set-java-field 1302
check-java-field 1302
java-field-class-name-for-setting 1302
record-java-class-lisp-symbol 1304
report-error-to-java-host 1305
reset-java-interface-for-new-jvm 1306
send-message-to-java-host 1307
setup-deliver-dynamic-library-for-java 1308
setup-field-accessor 1312
setup-java-caller 1313
setup-java-constructor 1313
setup-lisp-proxy 1315
standard-java-object 1315
throw-an-exception 1317
*to-java-host-stream* 1318
*to-java-host-stream-no-scroll* 1319
verify-java-caller 1320
verify-java-callers 1321
verify-lisp-proxy 1324
verify-lisp-proxies 1324
write-java-class-definitions-to-file 1326
write-java-class-definitions-to-stream 1326

42 Java classes and methods  1331
com.lispworks.LispCalls 1331
com.lispworks.LispCalls.callIntV 1332
com.lispworks.LispCalls.callIntA 1332
com.lispworks.LispCalls.callDoubleV 1332
com.lispworks.LispCalls.callDoubleA 1332
com.lispworks.LispCalls.callObjectV 1332
com.lispworks.LispCalls.callObjectA 1332
com.lispworks.LispCalls.callVoidV 1332
com.lispworks.LispCalls.callVoidA 1332
com.lispworks.LispCalls.checkLispSymbol 1334
com.lispworks.LispCalls.createLispProxy 1335
com.lispworks.LispCalls.waitForInitialization 1336

43 Android Java classes and methods  1339
com.lispworks.Manager 1339
com.lispworks.Manager.init 1341
com.lispworks.Manager.status 1343
com.lispworks.Manager.init_result_code 1344
com.lispworks.Manager.mInitErrorString 1346
com.lispworks.Manager.loadLibrary 1346
com.lispworks.Manager.LispErrorReporter 1347
com.lispworks.Manager.setErrorReporter 1347
com.lispworks.Manager.LispGuiErrorReporter 1347
com.lispworks.Manager.setGuiErrorReporter 1347
44 The MP Package 1359
allowing-block-interrupts 1359
any-other-process-non-internal-server-p 1361
barrier 1362
barrier-arriver-count 1362
barrier-block-and-wait 1363
barrier-change-count 1366
barrier-count 1367
barrier-disable 1368
barrier-enable 1368
barrier-name 1369
barrier-pass-through 1370
barrier-unblock 1370
barrier-wait 1372
change-process-priority 1375
condition-variable 1376
condition-variable-broadcast 1376
condition-variable-signal 1377
condition-variable-wait 1378
condition-variable-wait-count 1379
*current-process* 1380
current-process-block-interrupts 1380
current-process-in-cleanup-p 1381
current-process-kill 1382
current-process-pause 1383
current-process-send 1385
current-process-set-terminate-method 1386
current-process-unblock-interrupts 1387
debug-other-process 1388
*default-process-priority* 1389
ensure-process-cleanup 1389
find-process-from-name 1391
funcall-async 1392
funcall-async-list 1392
general-handle-event 1393
get-current-process 1394
get-process 1395
get-process-private-property 1396
initialize-multiprocessing 1397
*initial-processes* 1398
last-callback-on-thread 1399
list-all-processes 1400
lock 1401
lock-and-condition-variable-broadcast 1401
lock-and-condition-variable-signal 1402
lock-and-condition-variable-wait 1404
lock-locked-p 1406
lock-owned-by-current-process-p 1407
lock-recursive-p 1407
lock-recursively-locked-p 1408
lock-name 1409
lock-owner 1409
mailbox 1411
mailbox-count 1411
mailbox-empty-p 1412
mailbox-full-p 1413
mailbox-not-empty-p 1414
mailbox-peek 1414
mailbox-read 1415
mailbox-reader-process 1416
mailbox-send 1417
mailbox-send-limited 1418
mailbox-size 1420
mailbox-wait 1421
mailbox-wait-for-event 1422
Contents

*main-process* 1424
make-barrier 1424
make-condition-variable 1426
make-lock 1427
make-mailbox 1429
make-named-timer 1430
make-semaphore 1431
make-timer 1432
map-all-processes 1433
map-all-processes-backtrace 1433
map-process-backtrace 1434
map-processes 1435
notice-fd 1436
process-alive-p 1436
process-all-events 1437
process-allow-scheduling 1438
process-arrest-reasons 1438
process-break 1439
process-continue 1439
process-exclusive-lock 1440
process-exclusive-unlock 1441
process-idle-time 1442
*process-initial-bindings* 1442
process-interrupt 1444
process-interrupt-list 1446
process-join 1447
process-kill 1448
process-lock 1449
process-mailbox 1450
process-name 1450
process-p 1451
process-plist 1451
process-poke 1452
process-priority 1454
process-private-property 1455
process-property 1456
process-reset 1457
process-run-function 1458
process-run-reasons 1461
process-run-time 1462
process-send 1463
process-sharing-lock 1465
process-sharing-unlock 1466
process-stop 1467
process-stopped-p 1468
process-terminate 1468
process-unlock 1470
process-unstop 1471
process-wait 1472
process-wait-for-event 1473
process-wait-function 1474
process-wait-local 1474
process-wait-local-with-periodic-checks 1477
process-wait-local-with-timeout 1478
process-wait-local-with-timeout-and-periodic-checks 1479
process-wait-with-timeout 1480
process-whostate 1482
processes-count 1482
pushnew-to-process-private-property 1483
pushnew-to-process-property 1484
ps 1485
remove-from-process-private-property 1486
remove-from-process-property 1486
remove-process-private-property 1487
remove-process-property 1488
schedule-timer 1489
schedule-timer-milliseconds 1491
schedule-timer-relative 1493
schedule-timer-relative-milliseconds 1494
semaphore 1496
semaphore-acquire 1496
semaphore-count 1498
semaphore-name 1498
semaphore-release 1499
semaphore-wait-count 1500
set-funcall-async-limit 1500
simple-lock-and-condition-variable-wait 1501
symeval-in-process 1503
timer-expired-p 1504
Contents

timer-name 1505
unnotice-fd 1506
unschedule-timer 1506
wait-processing-events 1508
with-exclusive-lock 1509
with-interrupts-blocked 1510
with-lock 1510
with-sharing-lock 1511
without-interrupts 1512
without-preemption 1513
yield 1514

45 The PARSERGEN Package 1515
defparser 1515

46 The SERIAL-PORT Package 1517
open-serial-port 1517
close-serial-port 1519
get-serial-port-state 1520
serial-port 1520
read-serial-port-char 1521
read-serial-port-string 1521
serial-port-input-available-p 1522
set-serial-port-state 1523
wait-serial-port-state 1524
write-serial-port-char 1524
write-serial-port-string 1525

47 The SQL Package 1527
accepts-n-syntax 1527
add-sql-stream 1528
attribute-type 1529
cache-table-queries 1531
*cache-table-queries-default* 1532
commit 1532
connect 1533
*connect-if-exists* 1541
connected-databases 1542
create-index 1542
create-table 1543
create-view 1545
create-view-from-class 1546
database-name 1546
decode-to-db-standard-date 1547
decode-to-db-standard-timestamp 1547
encode-db-standard-date 1548
encode-db-standard-timestamp 1548
*default-database* 1549
*default-database-type* 1550
*default-update-objects-max-len* 1550
def-view-class 1551
delete-instance-records 1558
delete-records 1559
delete-sql-stream 1560
destroy-prepared-statement 1561
disable-sql-reader-syntax 1561
disconnect 1562
do-query 1563
drop-index 1564
drop-table 1565
drop-view 1565
drop-view-from-class 1566
enable-sql-reader-syntax 1567
execute-command 1567
find-database 1568
initialize-database-type 1569
*initialized-database-types* 1570
insert-records 1570
instance-refreshed 1572
list-attribute-types 1573
list-attributes 1574
list-classes 1575
list-sql-streams 1576
list-tables 1576
lob-stream 1577
locally-disable-sql-reader-syntax 1578
locally-enable-sql-reader-syntax 1579
loop 1580
map-query 1582
Contents

*mysql-library-directories* 1583
*mysql-library-path* 1584
*mysql-library-sub-directories* 1585
ora-lob-append 1586
ora-lob-assign 1587
ora-lob-char-set-form 1587
ora-lob-char-set-id 1588
ora-lob-close 1589
ora-lob-copy 1590
ora-lob-create-empty 1591
ora-lob-create-temporary 1592
ora-lob-disable-buffering 1593
ora-lob-element-type 1594
ora-lob-enable-buffering 1595
ora-lob-env-handle 1596
ora-lob-erase 1596
ora-lob-file-close 1597
ora-lob-file-close-all 1598
ora-lob-file-exists 1599
ora-lob-file-get-name 1600
ora-lob-file-is-open 1600
ora-lob-file-open 1601
ora-lob-file-set-name 1602
ora-lob-flush-buffer 1603
ora-lob-free 1604
ora-lob-free-temporary 1605
ora-lob-get-buffer 1605
ora-lob-get-chunk-size 1608
ora-lob-get-length 1608
ora-lob-internal-lob-p 1609
ora-lob-is-equal 1610
ora-lob-is-open 1610
ora-lob-is-temporary 1611
ora-lob-load-from-file 1612
ora-lob-lob-locator 1613
ora-lob-locator-is-init 1614
ora-lob-open 1614
ora-lob-read-buffer 1615
ora-lob-read-into-plain-file 1618
ora-lob-read-foreign-buffer 1619
ora-lob-svc-ctx-handle 1620
ora-lob-trim 1620
ora-lob-write-buffer 1621
ora-lob-write-from-plain-file 1623
ora-lob-write-foreign-buffer 1624
p-oci-env 1625
p-oci-file 1626
p-oci-lob-locator 1626
p-oci-lob-or-file 1626
p-oci-svc-ctx 1626
prepare-statement 1627
prepared-statement 1629
print-query 1629
query 1631
reconnect 1632
restore-sql-reader-syntax-state 1633
rollback 1634
select 1634
set-prepared-statement-variables 1639
simple-do-query 1640
sql 1642
sql-connection-error 1643
sql-database-data-error 1643
sql-database-error 1643
"sql-enlarge-static" 1644
sql-expression 1645
sql-expression-object 1647
sql-fatal-error 1648
sqlite-blob 1648
sqlite-last-insert-rowid 1649
sqlite-open-blob 1650
sqlite-close-blob 1650
sqlite-blob-p 1650
sqlite-blob-length 1650
replace-from-sqlite-blob 1650
replace-into-sqlite-blob 1650
sqlite-reopen-blob 1650
sqlite-raw-blob 1654
sqlite-raw-blob-p 1655
sqlite-raw-blob-valid-p 1655
48 The STREAM Package  1681

buffered-stream 1681
fundamental-binary-input-stream 1683
fundamental-binary-output-stream 1684
fundamental-binary-stream 1684
fundamental-character-input-stream 1685
fundamental-character-output-stream 1686
fundamental-character-stream 1687
fundamental-input-stream 1687
Contents

atomic-incf 1718
atomic-exchange 1719
atomic-fixnum-decf 1719
atomic-fixnum-incf 1719
atomic-pop 1720
atomic-push 1721
augmented-string 1722
simple-augmented-string 1722
augmented-string-p 1722
simple-augmented-string-p 1722
*binary-file-type* 1723
*binary-file-types* 1723
call-system 1724
call-system-showing-output 1727
cdr-assoc 1730
*check-network-server* 1731
coerce-to-gesture-spec 1731
compare-and-swap 1733
copy-preferences-from-older-version 1734
count-gen-num-allocation 1735
*debug-initialization-errors-in-snap-shot* 1736
default-eol-style 1737
*default-stack-group-list-length* 1738
define-atomic-modify-macro 1738
define-top-loop-command 1740
detect-eol-style 1742
detect-japanese-encoding-in-file 1743
detect-unicode-bom 1744
detect-utf32-bom 1744
detect-utf8-bom 1744
*directory-link-transparency* 1745
ensure-loads-after-loads 1746
ensure-memory-after-store 1746
ensure-stores-after-memory 1747
ensure-stores-after-stores 1748
*extended-spaces* 1748
*file-encoding-detection-algorithm* 1749
file-encoding-resolution-error 1751
*file-eol-style-detection-algorithm* 1751
*filename-pattern-encoding-matches* 1752
Contents

find-encoding-option 1752
find-filename-pattern-encoding-match 1753
gen-num-segments-fragmentation-state 1754
generation-number 1755
gesture-spec-accelerator-bit 1756
gesture-spec-caps-lock-bit 1756
gesture-spec-control-bit 1756
gesture-spec-hyper-bit 1756
gesture-spec-meta-bit 1756
gesture-spec-shift-bit 1756
gesture-spec-super-bit 1756
gesture-spec-data 1758
gesture-spec-modifiers 1758
gesture-spec-p 1759
gesture-spec-to-character 1760
get-file-stat 1761
get-folder-path 1762
get-maximum-allocated-in-generation-2-after-gc 1765
get-user-profile-directory 1767
globally-accessible 1768
guess-external-format 1769
immediatep 1770
in-static-area 1771
int32 1772
int32* 1773
int32+ 1773
int32- 1773
int32/= 1775
int32< 1775
int32<= 1775
int32= 1775
int32> 1775
int32>= 1775
int32<< 1776
int32>> 1776
Contents

int32-aref 1777
int32-logand 1778
int32-logandc1 1778
int32-logandc2 1778
int32-logbitp 1778
int32-logeqv 1778
int32-logior 1778
int32-lognand 1778
int32-lognor 1778
int32-lognot 1778
int32-logorc1 1778
int32-logorc2 1778
int32-logtest 1778
int32-logxor 1778
int32-minusp 1781
int32-plusp 1781
int32-zerop 1781
int32-to-int64 1781
int32-to-integer 1782
int64 1783
int64* 1784
int64+ 1784
int64- 1784
int64/ 1784
+int64-0+ 1785
+int64-1+ 1785
int64-1+ 1785
int64-1- 1785
int64/= 1786
int64/= 1786
int64< 1786
int64<= 1786
int64= 1786
int64> 1786
int64>= 1786
int64<< 1787
int64>> 1787
int64-aref 1788
int64-logand 1789
int64-logandc1 1789
int64-logandc2 1789
int64-logbitp 1789
int64-logeqv 1789
int64-logior 1789
int64-lognand 1789
int64-lognor 1789
int64-lognot 1789
int64-logorc1 1789
int64-logorc2 1789
int64-logtest 1789
int64-logxor 1789
int64-minusp 1791
int64-plusp 1791
int64-zerop 1791
int64-to-int32 1792
int64-to-integer 1793
integer-to-int32 1793
integer-to-int64 1794
*line-arguments-list* 1795
load-data-file 1795
locale-file-encoding 1796
low-level-atomic-place-p 1797
make-current-allocation-permanent 1798
make-gesture-spec 1800
make-object-permanent 1805
make-permanent-simple-vector 1807
make-simple-int32-vector 1808
make-simple-int64-vector 1809
make-stderr-stream 1810
make-typed-aref-vector 1811
map-environment 1812
marking-gc 1813
memory-growth-margin 1815
merge-ef-specs 1816
mobile-gc-p 1817
mobile-gc-sweep-objects 1818
object-address 1819
object-pointer 1820
octet-ref 1821
base-char-ref 1821
open-pipe 1823
Contents

open-url 1827
package-flagged-p 1827
pipe-close-connection 1828
pipe-exit-status 1829
pipe-kill-process 1830
pointer-from-address 1832
print-pretty-gesture-spec 1833
*print-symbols-using-bars* 1834
product-registry-path 1835
release-object-and-nullify 1837
room-values 1838
run-shell-command 1839
safe-locale-file-encoding 1844
set-approaching-memory-limit-callback 1845
set-automatic-gc-callback 1846
set-blocking-gen-num 1848
set-default-segment-size 1851
set-delay-promotion 1852
set-expected-allocation-in-generation-2-after-gc 1853
set-file-dates 1857
set-generation-2-gc-options 1858
set-gen-num-gc-threshold 1860
set-maximum-memory 1862
set-maximum-segment-size 1863
set-memory-check 1865
set-memory-exhausted-callback 1866
set-promote-generation-1 1868
set-reserved-memory-policy 1869
set-signal-handler 1870
set-spare-keeping-policy 1871
set-split-promotion 1872
set-static-segment-size 1874
set-temp-directory 1875
setup-atomic-fncall 1876
*sg-default-size* 1877
simple-int32-vector 1878
simple-int32-vector-length 1879
simple-int32-vector-p 1879
simple-int64-vector 1880
simple-int64-vector-length 1880
simple-int64-vector-p 1881
*specific-valid-file-encodings* 1881
specific-valid-file-encoding 1882
*stack-overflow-behaviour* 1883
staticp 1884
storage-exhausted 1884
sweep-gen-num-objects 1885
typed-aref 1886
wait-for-input-streams 1888
wait-for-input-streams-returning-first 1889
with-modification-change 1890
with-modification-check-macro 1891
with-other-threads-disabled 1892

50 Miscellaneous WIN32 symbols 1895

canonicalize-sid-string 1895
connect-to-named-pipe 1896
dismiss-splash-screen 1898
impersonating-named-pipe-client 1899
impersonating-user 1900
known-sid-integer-to-sid-string 1901
*latin-1-code-pages* 1902
long-namestring 1903
*multibyte-code-page-ef* 1904
named-pipe-stream-name 1904
open-named-pipe-stream 1905
record-message-in-windows-event-log 1909
security-description-string-for-open-named-pipe 1910
set-application-themed 1914
short-namestring 1915
sid-string-to-user-name 1916
str 1916
lpcstr 1916
lpstr 1916
tstr 1917
lpcstr 1917
lptstr 1917
wstr 1918
lpcwstr 1918
Contents

lpwstr 1918
user-name-to-sid-string 1919
wait-for-connection 1919
with-windows-event-log-event-source 1921

51 The Windows registry API 1923

close-registry-key 1923
collect-registry-subkeys 1924
collect-registry-values 1925
create-registry-key 1927
delete-registry-key 1929
elem-registry-value 1929
open-registry-key 1932
query-registry-key-info 1933
query-registry-value 1934
registry-key-exists-p 1935
registry-value 1935
set-registry-value 1937
with-registry-key 1938

52 The DDE client interface 1941

dde-advise-start 1941
dde-advise-start* 1944
dde-advise-stop 1945
dde-advise-stop* 1947
dde-client-advise-data 1948
dde-connect 1949
dde-disconnect 1950
dde-execute 1950
dde-execute* 1951
dde-execute-command 1952
dde-execute-command* 1953
dde-execute-string 1955
dde-execute-string* 1955
dde-item 1957
dde-item* 1958
dde-poke 1960
dde-poke* 1962
dde-request 1964
dde-request 1966
define-dde-client 1967
with-dde-conversation 1968

53 The DDE server interface 1971

dde-server-poke 1971
dde-server-request 1972
dde-server-topic 1973
dde-server-topics 1974
dde-system-topic 1975
dde-topic 1975
dde-topic-items 1976
define-dde-dispatch-topic 1977
define-dde-server 1978
define-dde-server-function 1979
start-dde-server 1983

54 Dynamic library C functions 1985

InitLispWorks 1985
LispWorksDlsym 1988
LispWorksState 1989
SimpleInitLispWorks 1990
QuitLispWorks 1991

Index 1993
About this manual

This manual contains a user guide section (previously published separately as the *LispWorks User Guide*) and a reference section (previously the *LispWorks Reference Manual*).

User Guide section

The user guide section of this manual describes the main language-level features and tools available in LispWorks, and how to use them.

These chapters describe the central programming tools and features in LispWorks:

- Chapter 1, “Starting LispWorks” describes how to start LispWorks and supply command line arguments.
- Chapter 2, “The Listener” describes the read-eval-print loop (REPL) listener.
- Chapter 3, “The Debugger” describes the REPL debugger.
- Chapter 4, “The REPL Inspector” describes the REPL inspector.
- Chapter 5, “The Trace Facility” describes the tracer.
- Chapter 6, “The Advice Facility”.

Preface
• Chapter 7, “Dspecs: Tools for Handling Definitions” describes the naming system for Lisp definitions, and in particular how to locate these.
• Chapter 8, “Action Lists” describes how you can run code at various hook points.
• Chapter 9, “The Compiler” describes the compiler optimization qualities and some ways to optimize your code.
• Chapter 10, “Code Coverage” shows you how to determine and visualize which parts of your program have actually run.
• Chapter 11, “Memory Management” covers the behavior (and for wizard level users, configuration) of the garbage collector.
• Chapter 12, “The Profiler” describes a tool for identifying bottlenecks impeding performance of your program.

The next chapter, Chapter 13, “Customization of LispWorks”, explains how to perform some commonly required customizations, such as controlling start-up appearance of LispWorks.

The remaining user guide chapters describe features of specialist interest:
• Chapter 14, “LispWorks as a dynamic library” describes how LispWorks operates as a DLL, .dylib or .so.
• Chapter 15, “Java interface” describes the LispWorks Java interface.
• Chapter 16, “Android interface” describes the LispWorks Android interface, which allows you to include a LispWorks runtime in an Android app.
• Chapter 17, “iOS interface” describes the LispWorks iOS interface, which allows you to include a LispWorks runtime in an iOS app.
• Chapter 18, “The Metaobject Protocol” describes how the LispWorks MOP implementation differs from AMOP.
• Chapter 19, “Multiprocessing”, including locks.
• Chapter 20, “Common Defsystem and ASDF” describes how to use defsystem to combine a series of source files into a manageable project.
• Chapter 21, “The Parser Generator”.
Preface

- Chapter 22, “Dynamic Data Exchange” describes how to implement DDE functionality in your Microsoft Windows applications.
- Chapter 23, “Common SQL” explains how to use LispWorks to communicate with databases using SQL.
- Chapter 24, “User Defined Streams” provides an illustrative example showing how to define and implement your own streams.
- Chapter 25, “TCP and UDP socket communication and SSL” describes the use of socket streams, including the Secure Sockets Layer (SSL).
- Chapter 26, “Internationalization: characters, strings and encodings” provides an overview of using international characters.
- Chapter 27, “LispWorks’ Operating Environment” explains how to find information about the Operating System and how LispWorks was started.
- Chapter 28, “Miscellaneous Utilities” describes miscellaneous functionality which does not belong in other chapters.
- Chapter 29, “64-bit LispWorks” outlines differences between 64-bit LispWorks and 32-bit LispWorks.
- Chapter 30, “Self-contained examples” enumerates the example files which are relevant to the content of this manual and are available in the LispWorks library.


Reference section

Most of the reference section is organized by package: each chapter contains reference material for the exported symbols in a given package. The chapters are organized alphabetically by package name.

Generally one chapter covers each package, but the WIN32 package symbols are split into four chapters, and the last chapter contains reference material for C functions. Within each chapter, the symbols are organized alphabetically (ignoring non-alphanumeric characters that are common in Lisp symbols, such as *). The chapters are:
Preface

- Chapter 31, “The CLOS Package”, describes the LispWorks extensions to CLOS, the Common Lisp Object System.
- Chapter 32, “The COMM Package”, describes the functions providing the TCP/IP interface.
- Chapter 33, “The COMMON-LISP Package”, describes the LispWorks extensions to symbols in the COMMON-LISP package. You should refer to the Common Lisp HyperSpec, supplied in HTML format with LispWorks, for full documentation about standard Common Lisp symbols.
- Chapter 34, “The COMPILER Package”, describes symbols available in the COMPILER package.
- Chapter 35, “The DBG Package”, describes symbols available in the DBG package, used to configure the debugging information produced by LispWorks.
- Chapter 36, “The DSPEC Package”, describes the symbols available in the DSPEC package, which are used for naming and locating definitions.
- Chapter 37, “The EXTERNAL-FORMAT Package”, describes symbols available in the EXTERNAL-FORMAT package.
- Chapter 38, “The HCL Package”, describes symbols available in the HCL package.
- Chapter 40, “The LISPWORKS Package”, describes symbols available in the LISPWORKS package.
- Chapter 41, “The LW-JI Package”, describes symbols available in the LW-JI package, which allows you to call to and from Java. This chapter describes the Java classes and methods available in LispWorks.
- Chapter 42, “Java classes and methods” describes the Java classes and methods available in LispWorks.
- Chapter 43, “Android Java classes and methods” describes the additional Java classes and methods available in LispWorks for Android Runtime.
Preface

- Chapter 44, “The MP Package”, describes symbols available in the **MP** package, giving you access to the multiprocessing capabilities of LispWorks.
- Chapter 45, “The PARSERGEN Package”, describes symbols available in the **PARSERGEN** package, the LispWorks parser generator.
- Chapter 46, “The SERIAL-PORT Package” documents the Serial Port API. This is implemented only in LispWorks for Windows.
- Chapter 47, “The SQL Package” documents symbols used in accessing LispWorks ODBC and SQL functionality.
- Chapter 48, “The STREAM Package” documents the symbols available in the **STREAM** package that provide users with the functionality to define their own streams for use by the standard I/O functions.
- Chapter 49, “The SYSTEM Package”, describes symbols available in the **SYSTEM** package.
- Chapter 50, “Miscellaneous WIN32 symbols”, describes miscellaneous symbols available in the **WIN32** package. It applies only to LispWorks for Windows.
- Chapter 51, “The Windows registry API”, describes the Windows registry API. It applies only to LispWorks for Windows.
- Chapter 52, “The DDE client interface”, describes the Dynamic Data Exchange (DDE) client API. It applies only to LispWorks for Windows.
- Chapter 53, “The DDE server interface”, describes the Dynamic Data Exchange (DDE) server API. It applies only to LispWorks for Windows.

Many of these reference chapters should be used in conjunction with corresponding chapters in the user guide section. Reference material for some aspects of LispWorks can be found in other manuals.

**Conventions used for reference entries**

Each entry is headed by the symbol name and type, followed by a number of fields providing further details. These fields consist of a subset of the follow-

Some symbols with closely-related functionality are coalesced into a single reference entry.

Throughout, variable arguments, slots and return values are italicised. They look like this.

Throughout, exported symbols and example code are printed like this. The package qualifier is usually omitted, unless the symbol is not documented in this manual.

Entries with a long “Description” section usually have as their first field a short “Summary” providing a quick overview of the symbol’s purpose.

The “Package” section shows the package from which the symbol is exported.

The “Signature” section shows the arguments and return values of functions and macros, and the parameters of types.

In a Generic Function entry there may be a “Method signatures” section showing system-defined method signatures.

The “Arguments” and “Values” sections show types of the arguments and return values.

In a Variable entry, the “Initial value” section shows the initial value.

In a Class entry the “Subclasses” section of lists the external subclasses, though not subclasses of those, and the “Superclasses” section lists the external superclasses, though not superclasses of those. The “Initargs” section describes the initialization arguments of the class, though note that initargs of superclasses are also valid. There may be an “Accessors” section listing accessor functions which are both readers and writers, and/or a “Readers” section listing accessor functions which are only readers. Accessor functions access the slot with matching name.

The “Description” section contains the detail of what the symbol does, how each argument is interpreted (and its default value if applicable), and how
each return value is derived. More incidental information may be shown in a “Notes” section.

A few entries have a “Compatibility notes” section describing changes in the symbol’s functionality relative to other LispWorks versions.

Examples are given under the “Examples” heading. Short examples are shown directly. Longer examples are supplied as source files in your LispWorks installation directory under examples/. The convenience function example-edit-file allows you to open these files in the LispWorks editor. The examples files are in a read-only directory and therefore you should compile them inside the IDE (by the Editor command Compile Buffer or the toolbar button or by choosing Buffer > Compile from the context menu), so it does not try to write a fasl file. If you want to manipulate an example file or compile it on the disk rather than in the IDE, then you need first to copy the file elsewhere (most easily by using the Editor command Write File or by choosing File > Save As from the context menu).

Finally, the “See also” section provides links to other related symbols and user guide sections.

The LispWorks manuals

The LispWorks manual set comprises the following books:

- The Common Lisp HyperSpec contains the specification for Common Lisp itself.
- The LispWorks User Guide and Reference Manual—this book—describes the main language-level features and tools available in LispWorks, along with an extensive reference of the functions, macros, variables and classes organized by package. Where LispWorks extends the functionality of a Common Lisp symbol, this is mentioned in Chapter 33, “The COMMON-LISP Package”
- The LispWorks IDE User Guide describes the LispWorks IDE, the user interface for LispWorks. This is a set of windowing tools that let you develop and test Common Lisp code more easily and quickly.
- The LispWorks Editor User Guide describes the keyboard commands and programming interface to the LispWorks IDE editor tool.
The CAPI User Guide and Reference Manual describes the CAPI. This is a library of classes, functions, and macros for developing graphical user interfaces for your applications. It comprises a tutorial guide to the CAPI and an in-depth reference text.


The LispWorks Delivery User Guide describes how you can deliver working, standalone versions of your LispWorks applications for distribution to your customers.

Developing Component Software with CORBA describes how LispWorks can interoperate with other CORBA-compliant systems.


The KnowledgeWorks and Prolog User Guide describes the LispWorks toolkit for building knowledge-based systems. Common Prolog is a logic programming system written in Common Lisp.

The LispWorks Release Notes and Installation Guide explains how to install LispWorks and start it running. It also contains Release Notes describing the new features in this release and any issues that could not be included in the other manuals.

The LispWorks manuals are all available in Portable Documentation Format (PDF). You can use Adobe Reader to browse the PDF documentation online or to print it. Adobe Reader is available for free download from Adobe’s web site at www.adobe.com.

The LispWorks manuals are also available in HTML format. Commands in the Help menu of any of the LispWorks IDE tools give you direct access to the HTML documentation, using your web browser. Details of how to use these commands can be found in the LispWorks IDE User Guide.
Please let us know if you find any mistakes in the LispWorks documentation, or if you have any suggestions for improvements.

Other documentation

The LispWorks manuals do not attempt to describe Lisp itself. For definitive information on Common Lisp, including CLOS, consult the American National Standard X3.226 for Common Lisp. An HTML version of this document is supplied with LispWorks and can be accessed from the Help menu.

For information on CLOS, Sonya E. Keene’s book *Object-Oriented Programming in Common Lisp: A Programmers’ Guide* is very helpful. This book is published by Addison-Wesley.

For an account of Metaobject protocols as well as a detailed study of an implementation of CLOS see Kiczales, des Rivières and Bobrow, *The Art of the Meta-Object Protocol*, published by MIT Press, often referred to as AMOP. The LispWorks MOP mostly conforms to chapters 5 & 6 of AMOP; the differences are mentioned here in Chapter 18, “The Metaobject Protocol”.

Notation and conventions

The LispWorks manuals follow the notation used in *Common Lisp: the Language (2nd Edition)*.

This manual often refers to example files in the LispWorks library, like this:

```
(example-edit-file "ssl/ssl-client")
```

These examples are Lisp source files in your LispWorks installation under `lib/7-1-0-0/examples/`. You can simply evaluate the given form to view the example source file.

Other references such as ”... the LispWorks file `foo/bar.lisp” mean a file `bar.lisp` in a subdirectory `foo` of the LispWorks library directory. Evaluate this form in your LispWorks image to obtain the full path of such a file:

```
(sys:lispworks-file "foo/bar.lisp")
```
Preface
Starting LispWorks

Firstly you need LispWorks installed as described in the Release Notes and Installation Guide.

1.1 The usual way to start LispWorks

On Microsoft Windows and Mac OS X the simplest way to run LispWorks is that provided in the desktop environment. On Windows you can run LispWorks from the desktop Start menu, or the Start screen on Windows 8. On Mac OS X you can run LispWorks by clicking on the "LW" icon in the Dock. On both these platforms you can create a shortcut to LispWorks and place it somewhere that is convenient for you, such as the Windows 8 taskbar.

On non-Windows and non-Mac OS X systems you start LispWorks by entering the name of the LispWorks executable at a shell prompt.

1.2 Passing arguments to LispWorks

Occasionally you may need to start LispWorks with certain arguments. This section describes the most frequent of these occasions.
1.2.1 Saving a new image

Note: If you use the LispWorks IDE, you may find a saved session more convenient than saving an image as described in this section. See “Saved sessions” on page 182 for more information.

To save a new image "by hand", create a suitable file `save-config.lisp` as described in the section "Saving and testing the configured image" in the LispWorks Release Notes and Installation Guide. Such a file should load any desired configuration, modules and application code, and lastly call `save-image`.

Then you run LispWorks with a command line which passes your file as a build script.

On Mac OS X, run Terminal.app to get a shell, and enter a line like this at the prompt:

```
% lispworks-7-1-0-x86-darwin -build /tmp/save-config.lisp
```

On Microsoft Windows, run Command Prompt to get a DOS shell, and enter a line like this:

```
C:\Program Files\LispWorks>lispworks-7-1-0-x86-win32.exe -build
C:\temp\save-config.lisp
```

On Linux, get a shell and enter a line like this:

```
% lispworks-7-1-0-x86-linux -build /tmp/save-config.lisp
```

On UNIX, get a shell and enter a line like this:

```
% lispworks-7-1-0-sparc-solaris -build /tmp/save-config.lisp
```

When the command exits, a new image has been saved. You can run this new image directly from the command line, or create a shortcut or symbolic link to make it convenient to run.

With all the command lines above, if you perform the task frequently, make a script or a shortcut containing the command line, and run that.

Note that `save-config.lisp` no longer needs to do `(load-all-patches)` because `-build` calls `load-all-patches` automatically in LispWorks 6.1 and later versions. However, if `save-config.lisp` does call `load-all-patches`, this is harmless.
1.2.2 Saving a console mode image

To save a LispWorks image which does not start the LispWorks IDE by default, make a script similar to `save-config.lisp` above, but where you call

```
(save-image "my-console-lispworks" :environment nil)
```

The resulting new image, `my-console-lispworks`, can be made to start the LispWorks IDE either by calling `env:start-environment` or by passing `-env` or `-environment` on the command line.

1.2.3 Bypassing initialization files

If you do not want to load your personal initialization file, for example to discover if the behavior of LispWorks is due to some setting of yours, pass `-init -` on the command line.

To start LispWorks without loading either the personal or site initialization files, start it like this:

```
lispworks -init -siteinit -
```

1.2.4 Other command line options

Other less commonly-used LispWorks command line arguments are described in “The Command Line” on page 454

1.3 Starting the LispWorks Graphical IDE

In LispWorks images shipped on the Windows, Mac OS X, Linux, x86/x64 Solaris, FreeBSD and AIX platforms, the IDE starts automatically by default. If you have an image saved such that the IDE does not start by default, you can start the IDE by calling the function `env:start-environment`. Such an image is shipped for UNIX platforms.

1.4 Using LispWorks with SLIME

Download SLIME from `http://common-lisp.net/project/slime/`
1.4.1 Using the Professional/Enterprise Editions with SLIME

To use LispWorks with SLIME it is best to use an image which does not start the LispWorks IDE automatically. You can create such an image with LispWorks Professional or Enterprise Edition. Save it as ~/_lw-console as described in “Saving a non-GUI image with multiprocessing enabled” on page 181.

Configure Emacs to use "-/lw-console" as the value of inferior-lisp-program as shown in the SLIME README.

1.4.2 Using the Personal Edition with SLIME

Start LispWorks Personal Edition, which starts the LispWorks IDE automatically.

Execute these forms in the LispWorks IDE:

```lisp
(load "/path/to/slime/swank-loader")
(swank-loader:init)
(swank:create-server :port 4005)
```

Inside Emacs, Meta+X slime-connect. Use the same port given above.

1.5 Quitting LispWorks

To quit LispWorks from the LispWorks IDE, use one of the following:

- The menu command File > Exit all platforms except Mac OS X.
- The menu command LispWorks > Quit LispWorks on Mac OS X.
- The key Command+Q on Mac OS X
- The key sequence Ctrl+X Ctrl+C in an editor-based tool such as the Editor or Listener
- A platform/window-manager-specific exit gesture such as clicking a close button on the Podium window
- Call the function quit.

To quit LispWorks when running in console mode or via SLIME, simply call quit.
The Listener

The listener is another name for the read-eval-print loop (REPL) which allows you to interactively evaluate Lisp forms and see their output and return values. Lisp programmers typically do incremental development and testing in a listener before saving the working code to disk.

This chapter describes the basic use of a LispWorks listener. You might access this in a terminal (Unix shell) or MS-DOS command window. Alternatively the LispWorks IDE contains a graphical Listener tool which runs a REPL and supports all the functionality described in this chapter, as well as its own graphical features. Please refer to the LispWorks IDE User Guide for details specific to the graphical Listener tool.

2.1 First use of the listener

LispWorks runs a top-level REPL on startup. The listener by default appears with a prompt. The name of the current package (that is, the value of cl:*package*) is printed followed by a positive integer, like this:

    CL-USER 1 >

Enter a Lisp form after the prompt and press Return:
The first ’42’ printed is the output of the call to `print`. You see it here because output sent to `*standard-output*` is written to the listener. The second ’42’ printed is the return value of the call to print. After the return value a new prompt appears. Notice that it contains ‘2’ after the package name: your successive inputs are numbered. You can now proceed to develop and test pieces of your application code:

```
CL-USER 2 > (defstruct animal species name weight)
ANIMAL

CL-USER 3 > (make-animal :species "Hippopotamus" :name "Hilda" :weight 42)
#S(ANIMAL :SPECIES "Hippopotamus" :NAME "Hilda" :WEIGHT 42)
```

### 2.2 Standard listener commands

Generally the listener simply evaluates Lisp forms that you enter. However a few keywords, described in this section, are specially recognized as shortcut for common listener operations.

#### 2.2.1 Standard top-level loop commands

##### :redo

`Listener command`

```
:redo &optional command-identifier
```

This option repeats a previous input. The `command-identifier` is either a number in the listener’s history list or a symbol or subform in the input to repeat. If `command-identifier` is not supplied, the last input is repeated.
2.2 Standard listener commands

:get retrieves a previously-entered input from the listener’s history and places it in the variable name. The command-identifier is the history list number of the input to be retrieved.

:use Listener command

:use new old &optional command-identifier

:use does a variant of a previous input. old matches a symbol or subform in the previous input, and is replaced with new to construct the new input. If supplied, command-identifier is the history list number of the input you want to modify.

:his Listener command

:his &optional n m

:his produces a list of the input history. If n is supplied it should be a positive integer: the last n inputs are shown. If m is also supplied it should be a positive integer greater than n, when inputs numbered n through m in the history are shown.

:bug-form Listener command

:bug-form subject &key filename

:bug-form prints a template bug report suitable for sending to Lisp Support. Supply a string subject. If you also supply filename, the report is printed to the file.

:help Listener command

:help

:help prints a brief listing of the available listener commands.

:? Listener command

:?
The Listener

?: is a synonym for :help.

2.2.2 Examples

CL-USER 4 > :redo
(MAKE-ANIMAL :SPECIES "Hippopotamus" :NAME ...)
#S(ANIMAL :SPECIES "Hippopotamus" :NAME "Hilda" :WEIGHT 42)

CL-USER 5 > :his

1: (PRINT 42)
2: (DEFSTRUCT ANIMAL SPECIES NAME ...)
3: (MAKE-ANIMAL :SPECIES "Hippopotamus" :NAME ...)
4: (MAKE-ANIMAL :SPECIES "Hippopotamus" :NAME ...)

CL-USER 5 > :get make-hilda 3

CL-USER 5 > make-hilda
(MAKE-ANIMAL :SPECIES "Hippopotamus" :NAME "Hilda" :WEIGHT 42)

CL-USER 6 > :use "Henry" "Hilda"
(MAKE-ANIMAL :SPECIES "Hippopotamus" :NAME ...)
#S(ANIMAL :SPECIES "Hippopotamus" :NAME "Henry" :WEIGHT 42)

CL-USER 7 > :bug-form "Too many hippos..." :filename "bug-report.txt"

2.3 The listener prompt

The variable *prompt* controls the appearance of the listener prompt. See *prompt*, page 1163 if you want to alter this.

If the default prompt contains a colon followed by a second positive integer then you are no longer in the top-level loop, but have entered the REPL debugger, as described in “The Debugger” on page 9.
The debugger is an interactive tool for examining and manipulating the Lisp environment. Within the debugger you have access to not only the interpreter, but also to a variety of debugging tools. The default behavior when any error occurs is to enter the debugger. Users can then trace backwards through the history of function calls to determine how the error arose. They may inspect and alter local variables of the functions on the execution stack, and possibly continue execution by invoking a pre-defined restart (if available) or by forcing any function invocation on the stack to return user-specified values.

When writing an application it is possible to prevent entry to the debugger when an error occurs, by creating condition handlers to take some appropriate action to recover without user intervention. It is also possible to use restarts to specify some default methods of error recovery. The debugger is entered whenever an error is signaled (via a call to `error` or `cerror`) and not handled by an error handler, or it can be explicitly invoked via a call to `break`.

You can use the debugger in REPL mode (that is, in the listener read-eval-print loop) or using the graphical Debugger tool in the LispWorks IDE. This chapter describes the REPL debugger; please refer to the *LispWorks IDE User Guide* for details about the graphical Debugger tool.

The compiler generates information necessary for the use of the debugger during compilation. You can opt for faster compilation, at the expense of
reducing the information available to the debugger, using `toggle-source-debugging`.

### 3.1 Entering the REPL debugger

The following is a simple example.

```lisp
CL-USER 2 > (defun make-a-hippo (name weight)
           (if (numberp weight)
               (make-animal 'hippo name weight)
               (error "Argument to make-a-hippo not a number")))
MAKE-A-HIPPO

CL-USER 3 > (make-a-hippo "Hilda" nil)
Error: Argument to make-a-hippo not a number
1 (abort) Return to level 0.
2 Return to top loop level 0.

Type :b for backtrace or :c <option number> to proceed.
Type :bug-form "<subject>" for a bug report template or :? for other options.

CL-USER 4 : 1 >
```

The call to `error` causes entry into the debugger. The final prompt in the example contains a 1 to indicate that the top level of the debugger has been entered. The debugger can be entered recursively, and the prompt shows the current level. Once inside the debugger, you may use all the facilities available at the top-level in addition to the debugger commands.

The debugger may also be invoked by using the trace facility to force a break at entry to or exit from a particular function.

The debugger can also be entered by a keyboard interrupt. Keyboard interrupts are generated by the `break gesture`, which varies between the supported systems as follows:

- **Microsoft Windows**
  - `Ctrl+Break`

- **GTK and Motif** `Meta+Ctrl+C`
3.2 Simple use of the REPL debugger

Break if keyboard has that key. Note that PC keyboards do not have Break, only Ctrl+Break, which is different. See also capi:set-interactive-break-gestures.

Cocoa Command+Control+, (comma). This is only supported on Mac OS X 10.4 and newer.

When the break gesture is used, LispWorks attempts to find a busy process to break. If there is no obvious candidate and the LispWorks IDE is running, then it displays the Process Browser tool.

3.2 Simple use of the REPL debugger

Upon entering the debugger as a result of an error, a message describing the error is printed and a number of options to continue (called restarts) are presented. Thus:

CL-USER 6 > (/ 3 0)

Error: Division-by-zero caused by / of (3 0)
1 (continue) Return a value to use
2 Supply new arguments to use
3 (abort) return to level 0.
4 return to top loop level 0.
5 Destroy process.

Type :c followed by a number to proceed

CL-USER 7 : 1 >

To select one of these restarts, enter :c (continue) followed by the number of the restart. So in the above example you could continue as follows:

CL-USER 7 : 1 > :c 2

Supply first number: 33
Supply second number: 11
3

CL-USER 8 >
There are two special restarts, a continue restart and an abort restart. These are indicated by the bracketed word continue or abort at their start. The continue restart can be invoked by typing \texttt{:c} alone. Similarly, the abort restart can be invoked by entering \texttt{:a}. So an alternative continuation of the division example would be:

\begin{verbatim}
CL-USER 7 : 1 > :c

Supply a form to be evaluated and used: (+ 4 5)
\end{verbatim}

9

3.3 The stack in the debugger

The debugger allows you to examine the state of the execution stack. This consists of a sequence of frames representing active function invocations, special variable bindings, restarts, active catchers, active handlers and system-related code. In particular the execution stack has a call frame for each active function call (that is for each function that has been entered but from which control has not yet returned). The top of the stack contains the most recently created frames (and so the innermost calls), and the bottom of the stack contains the oldest frames (and so the outermost calls). You can examine a call frame to find the function’s name, and the names and values of its arguments.

The function call frames displayed are affected by any \texttt{hcl:alias} and \texttt{hcl:invisible-frame} declarations. See \texttt{declare}, page 651 for the details.

Catch frames are established by using the special form catch, and exist to receive throws to the matching tag. Restart frames correspond to restarts that have been set up, and handler frames correspond to the error handlers currently active. Binding frames are formed when special variables are bound. Open frames are established by the system. By default only the catch frames and the call frames are displayed. However the remaining types of frame are displayed if you set the appropriate variables (see Section 3.6 on page 23).

Within the debugger there are commands to examine a stack frame, and to move around the stack. These are explained in the following section. Typing \texttt{help} in the debugger also produces a command listing.

One of the most useful features is that you can access a local variable in the current frame simply by entering its name as shown in the backtrace. See step 7 in “Example debugging session” on page 21.
3.4 REPL debugger commands

This section describes commands specific to the debugger. In the debugger, you can also do anything that you can do in the top-level loop including evaluation of forms and the standard listener commands.

Upon entry to the debugger the implicit current stack frame is set to the top of the execution stack. The debugger commands allow you to move around the stack, to examine the current frame, and to leave the debugger. The commands are all keywords, and as such case-insensitive, but are shown here in lower case for clarity.

You can get brief help listing these commands by entering : ? at the debugger prompt.

3.4.1 Backtracing

A backtrace is a list of the stack frames starting at the current frame and continuing down the stack. The backtrace thus displays the sequence by which the functions were invoked, starting with the most recent. For instance:
In the above example the command to show a quick backtrace was used (:bq). Instead of showing each stack frame fully, this only shows the function name associated with each of the call frames. The number 10 following :bq specifies that only the next ten frames should be displayed rather than continuing to the bottom of the stack.

:bq

This is the command to obtain a backtrace from the current frame. It may optionally be followed by :verbose, in which case a fuller description of each frame is given that includes the values of the arguments to the
function calls. It may also be followed by a number \((m)\), specifying that only that number of frames should be displayed.

**:bq**

*Debugger command*

**:bq m**

This produces a quick backtrace from the current position. Only the call frames are included, and only the names of the associated functions are shown. If the command is followed by a number then only that many frames are displayed.

### 3.4.2 Moving around the stack

On entry to the debugger the current frame is the one at the top of the execution stack. There are commands to move to the top and bottom of the stack, to move up or down the stack by a certain number of frames, and to move to the frame representing an invocation of a particular function.

**:>**

*Debugger command*

This sets the current frame to the one at the bottom of the stack.

**:<**

*Debugger command*

This sets the current frame to the one at the top of the stack.

**:p**

*Debugger command*

**:p \([m|fn-name|fn-name-substring]\)**

By default this takes you to the previous frame on the stack. If it is followed by a number then it moves that number of frames up the stack. If it is followed by a function name then it moves to the previous call frame for that function. If it is followed by a string then it moves to the previous call frame whose function name contains that string.
3 The Debugger

: n  

Debugger command

: n  [m | fn-name | fn-name-substring]

Similar to the above, this goes to the next frame down the stack, or m frames down the stack, or to the next call frame for the function fn-name, or to the next call frame whose function name contains fn-name-substring.

3.4.3 Miscellaneous commands

:v  

Debugger command

This displays information about the current stack frame. In the case of a call frame corresponding to a compiled function the names and values of the function’s arguments are shown. Closure variables (either from an outer scope or used by an inner scope) and special variables are indicated by {Closing} or {Special} as in this session:
3.4 REPL debugger commands

CL-USER 32 > (compile (defun foo (*zero* one two)
   (declare (special *zero*))
   (list (/ one *zero*) #'(lambda () one)
   two)))
FOO
NIL
NIL

CL-USER 33 > (foo 0 1 2)
Error: Division-by-zero caused by / of (1 0).
  1 (continue) Return a value to use.
  2 Supply new arguments to use.
  3 (abort) Return to level 0.
  4 Return to top loop level 0.
Type :b for backtrace or :c <option number> to proceed.
Type :bug-form "<subject>" for a bug report template or :? for other options.

CL-USER 34 : 1 > :n foo
Call to FOO

CL-USER 35 : 1 > :v
Call to FOO {offset 114}
   *ZERO* {Special} : 0
   ONE {Closing}    : 1
   TWO               : 2

CL-USER 36 : 1 >
For an interpreted function the names and values of local variables are also shown.
If the value of an argument is not known (perhaps because the code has been compiled for speed rather than other considerations), then it is printed as the keyword :dont-know.

Debugger Command

:1 [m | var-name | var-name-substring]

By default this prints a list of the values of all the local variables in the current frame. If the command is followed by a number then it prints the value of the $m$th local variables (counting from 0, in the order shown
by the :v command). If it is followed by a variable name var-name then it prints the value of that variable (note that the same effect can be achieved by just entering the name of the variable into the Listener). If it is followed by a string var-name-substring then it prints the value of the first variable whose name contains var-name-substring.

In all cases, * is set to the printed value.

: error

Debugger command

This reprints the message which was displayed upon entry to the current level of the debugger. This is typically an error message and includes several continuation options.

: cc

Debugger command

: cc &optional var

This returns the current condition object which caused entry to this level of the debugger. If an optional var is supplied then this must be a symbol, whose symbol-value is set to the value of the condition object.

: ed

Debugger command

This allows you to edit the function associated with the current frame. If you are using TAGS, you are prompted for a TAGS file.

: all

Debugger command

: all &optional flag

This option enables you to set the debugger option to show all frames (if flag is non-nil), or back to the default (if flag is nil). By default, flag is t. See also set-debugger-options.
### 3.4 REPL debugger commands

**:lambda**

Debugger command

This returns the lambda expression for an anonymous interpreted frame. If the expression is not known, then it is printed as the keyword `:dont-know`.

**:func**

Debugger command

`:func &optional disassemble-p`

This returns (and sets * to) the function object of the current frame. This is especially useful for the call frame of functions that are not the symbol function of some symbols, for example closures and method functions.

If `disassemble-p` is true, `:func` first disassembles the function, and then returns it and sets *. The default value of `disassemble-p` is `nil`.

`:func` is applicable only in call frames.

**:lf**

Debugger command

This command prints symbols from other packages corresponding to the symbol that was called, but could not be found, in the current package. Any such symbols are also offered as restarts when you first enter the debugger.
NEW 21 > (initialize-graphics-port)

Error: Undefined function INITIALIZE-GRAPHICS-PORT called with arguments ().
1 (continue) Try invoking INITIALIZE-GRAPHICS-PORT again.
2 Return some values from the call to INITIALIZE-GRAPHICS-PORT.
3 Try invoking GRAPHICS-PORTS:INITIALIZE-GRAPHICS-PORT with the same arguments.
4 Set the symbol-function of INITIALIZE-GRAPHICS-PORT to the symbol-function of GRAPHICS-PORTS:INITIALIZE-GRAPHICS-PORT.
5 Try invoking something other than INITIALIZE-GRAPHICS-PORT with the same arguments.
6 Set the symbol-function of INITIALIZE-GRAPHICS-PORT to another function.
7 (abort) Return to level 0.
8 Return to top loop level 0.

Type :c followed by a number to proceed or type :? for other options

NEW 22 : 1 > :lf
Possible candidates are (GRAPHICS-PORTS:INITIALIZE-GRAPHICS-PORT)
GRAPHICS-PORTS:INITIALIZE-GRAPHICS-PORT

NEW 23 : 1 >

3.4.4 Leaving the debugger

You may leave the debugger either by taking one of the continuation options initially presented, or by explicitly specifying values to return from one of the frames on the stack.

:a
Debugger command

This selects the :abort option from the various continuation options that are displayed when you enter the current level of the debugger.

:c
Debugger command
:c &optional m

If this is followed by a number then it selects the option with that number, otherwise it selects the :continue option.
3.4 REPL debugger commands

:ret  
**Debugger command**

:ret value

This causes value to be returned from the current frame. It is only possible to use this command when the current frame is a call frame. Multiple values may be returned by using the values function. So to return the values 1 and 2 from the current call frame, you could type

:ret (values 1 2)

:res  
**Debugger command**

:res m

Restarts the current frame. If m is nil, you are prompted for new arguments which should be entered on one line, separated by whitespace. If m is true or is not supplied, the original arguments to the frame are used.

;top  
**Debugger command**

Aborts to the top level of the debugger. A synonym is :a :t.

3.4.5 Example debugging session

This section presents a short interactive debugging session. It starts by defining a routine to calculate Fibonacci Numbers, and then erroneously calls it with a string.

1. First, define the `fibonacci` function shown below in a listener.

   (defun fibonacci (m)
      (let ((fib-n-1 1)
            (fib-n-2 1)
            (index 2))
        (loop
          (if (= index m) (return fib-n-1))
          (incf index)
          (psetq fib-n-1 (+ fib-n-1 fib-n-2)
                       fib-n-2 fib-n-1)))

2. Next, call the function as follows.

   (fibonacci *turtle*)
The system generates an error, since `cl:=' expects its arguments to be numbers, and displays several continuation options, so that you can try to find out how the problem arose.

3. Enter `:bb` at the debugger prompt to obtain a full backtrace.

Notice that the problem is in the call to `fibonacci`.

You should have passed the length of the string as an argument to `fibonacci`, rather than the string itself.

4. Attempt to calculate this value now, by typing the following form at the debugger prompt.

```
(length "turtle")
```

You intended to call `fibonacci` with the length of the string, but entered `length` incorrectly. This takes you into the second level of the debugger. Note that the continuation options from your entry into the top level of the debugger are still displayed, and are listed after the new options. You can select any of these options.

5. Enter `:a` to abort one level of the debugger.

6. Enter `:error` to remind yourself of the original error that you need to handle. You need to fix the value passed as the second argument to `fibonacci`.

7. Enter `:n fibonacci` to move to the stack frame for the call to `fibonacci`.

8. Enter `:v` to display variable information about this stack frame:

```
Interpreted call to FIBONACCI:
 M       : "turtle"
 INDEX   : 3
 FIB-N-2 : 1
 FIB-N-1 : 2
```

You need to set the value of the variable `m` to be the length of the string "turtle", rather than the string itself.

9. Enter this form:

```
(setq m (length "turtle"))
```
3.5 Debugger troubleshooting

In order to get the original computation to resume using the new value of \( m \), you still need to handle the original error.

10. Enter `:error` to remind yourself of the original error condition.
    You can handle this error by returning `nil` from the call to `cl:=`, which is the result that would have been obtained if \( m \) had been correctly set initially.

11. Enter `:c` to invoke the `continue` restart, which in this case requires you to return a value to use.

12. When prompted for a form to be evaluated, enter `nil`.
    This causes execution to continue as desired. Notice that the correct result 8 is returned.

3.5 Debugger troubleshooting

Code which modifies the readtable case of the readtable can hinder debugger interaction. This is because standard commands entered as lowercase `:a` for example will not be recognized if the readtable case is `:preserve` or `:down-case`.

You can use `with-debugger-wrapper` together with `with-standard-io-syntax` to enable the debugger to recognize such input if the code in `body` should enter the debugger, like this:

```
(defun my-debugger-wrapper (func condition)
  (with-standard-io-syntax
    (funcall func condition)))

(dbg:with-debugger-wrapper
 'my-debugger-wrapper
 (let ((*readtable* (some-modified-readtable))
       (body)))
```

3.6 Debugger control variables

`cl:*debug-io*`

The value of this variable is the stream which the debugger uses for its input and output.

`dbg:*debug-print-length*`
The value to which \texttt{cl:*print-length*} is bound during output from the debugger.

\texttt{dbg:*debug-print-level*}

The value to which \texttt{cl:*print-level*} is bound during output from the debugger.

\texttt{dbg:*hidden-packages*}

This variable should be bound to a list of packages. The debugger suppresses symbols from these packages (so, for example, it does not display call frames for functions in these packages).

\texttt{dbg:*print-binding-frames*}

This variable controls whether binding frames are displayed by the debugger. The initial value is \texttt{nil}. The value can be set directly or by calling \texttt{set-debugger-options} which may be more convenient.

\texttt{dbg:*print-catch-frames*}

This variable controls whether catch frames are displayed by the debugger. The initial value is \texttt{nil}. The value can be set directly or by calling \texttt{set-debugger-options} which may be more convenient.

\texttt{dbg:*print-handler-frames*}

This variable controls whether handler frames are displayed by the debugger. The initial value is \texttt{nil}. The value can be set directly or by calling \texttt{set-debugger-options} which may be more convenient.

\texttt{dbg:*print-restart-frames*}

This variable controls whether restart frames are displayed by the debugger. The initial value is \texttt{nil}. The value can be set directly or by calling \texttt{set-debugger-options} which may be more convenient.
3.7 Remote debugging

Remote debugging allows you to debug a LispWorks process that is running on one machine using a LispWorks IDE that is running on another machine. It is intended to make it easier to debug applications running on machines that do not have the LispWorks IDE, mainly mobile device applications on iOS and Android, but also applications running on servers where you cannot run the LispWorks IDE.

In the discussion below, the process being debugged is referred to as the "client", and the process running the LispWorks IDE is referred to as the "IDE".

With remote debugging you can:

- Make the client, when it enters the debugger, open a GUI debugger in the IDE. The GUI debugger behaves like an ordinary GUI debugger, but the data it displays is from the client, and input into its Listener pane is read and evaluated by the client.

- Open a Listener in the IDE, where reading and evaluating input is done by the client. This can be done either by calling `start-remote-listener` on the client side, or `ide-open-a-listener` on the IDE side.

- Inspect remote objects, by using the Inspector as usual on the IDE side, or `remote-inspect` on the client side.

- Evaluate forms on the client side from the Editor on the IDE side.

When you look at the source code from an IDE tool that is displaying client side data (for example by using the Find Source menu item) or look at the class of a remote object, the IDE finds the matching source or class on its side. You need to ensure that the IDE and the client sides have the same sources and class definitions for that to work.

Remote debugging is based on "connections", which are implemented on top of streams connecting the two sides. In normal usage, LispWorks will open a TCP socket stream for a connection, but you can also create connections with your own streams or sockets.

A LispWorks process that has loaded the remote debugging module can be connected to several IDE processes simultaneously, and any IDE process can be connected to several clients. The same IDE process can act as the client side
and the IDE side at the same time. However, the most common usage is expected to be one client and one IDE, and the interface is designed towards this simple usage.

Communication across the connection is architecture-independent, and either side can be any architecture. It relies on there being a working Common Lisp reader.

The client side should load the client code by calling:

```
(require "remote-debugger-client")
```

Note that if the client is a delivered application, the call to `require` needs to happen at load time, before calling `deliver`. On the IDE side, the module "remote-debugger-full" (which includes "remote-debugger-client") is loaded automatically when needed.

### 3.7.1 Simple usage

In the simple usage scenario, you have one IDE and one client. To create the connection between them, you need to tell LispWorks how to create the TCP socket stream, which requires one side to be a TCP server, and the other side to have the address (or name) of the server-side machine to connect to. Therefore, in the simple case you will need to make a function call on both sides. Once you perform the two function calls, you can use most of the power of remote debugging.

There are two ways to specify the connection: one with the IDE acting as the TCP server, and one with the client acting as the TCP server.

#### 3.7.1.1 Using the IDE as the TCP server

On the IDE side, you should call:

```
(dbg:start-ide-remote-debugging-server)
```

On the client side, you should call:

```
(dbg:configure-remote-debugging-spec "ide-hostname")
```

After making these calls, whenever the debugger is entered on the client side, it will automatically display a GUI debugger on the IDE side. In addition, calls
3.7 Remote debugging

to `start-remote-listener` and `remote-inspect` from the client side will automatically display tools on the IDE side.

These functions use TCP port 21101 by default (the initial value of `*default-ide-remote-debugging-server-port*`).

By default, `configure-remote-debugging-spec` delays opening the connection until it is actually needed (by entering the debugger, or a call to `start-remote-listener` or `remote-inspect`).

Note that within the LispWorks IDE, you can make the call to `start-ide-remote-debugging-server` using the `Start IDE Remote Debugging Server` button in the `Preferences` dialog `Debugger` options `Remote` tab.

### 3.7.1.2 Using the client as the TCP server

On the client side you should call:

```
(dbg:start-client-remote-debugging-server)
```

On the IDE side you should call:

```
(dbg:ide-connect-remote-debugging "client-hostname")
```

The call on the IDE side opens a connection, which the client will use when entering the debugger and in calls to `start-remote-listener` and `remote-inspect`.

These functions use TCP port 21102 by default (the initial value of `*default-client-remote-debugging-server-port*`).

Note that within the LispWorks IDE, you can make the call to `ide-connect-remote-debugging` using the `Connect To Debugging Client` button in the `Preferences` dialog `Debugger` options `Remote` tab.

### 3.7.2 The client side of remote debugging

The client side remote debugging API is intended to minimize the amount of work you need to do for simple configurations.

Once you have either specified the connection by calling `configure-remote-debugging-spec` on the client side and called `start-ide-remote-debugging-server` on the IDE side, or called `start-client-remote-debugging-server`...
server on the client side and the IDE has connected to it using ide-connect-remote-debugging, entering the debugger automatically opens a Debugger window on the IDE side (unless you are already inside a Remote Listener or Remote Debugger).

If you want to open a Remote Listener on the IDE side from the client side, you can call start-remote-listener. Also, you can call remote-inspect on the client side to inspect an object on the IDE side.

The interface allows you to have more complex configurations, as detailed by configure-remote-debugging-spec, create-client-remote-debugging-connection and start-client-remote-debugging-server.

3.7.3 The IDE side of remote debugging

The behavior of the Debugger, Listener and Inspector tools is described in the LispWorks IDE User Guide.

Remote Debugger windows are opened automatically when the client side enters the debugger.

Remote Listener windows are opened on request, either by using the IDE's menus, by calling ide-open-a-listener, or by calling ide-connect-remote-debugging with :open-a-listener t (or from the client side by start-remote-listener).

The Inspector inspects a remote object when you tell it to inspect in the same way as you would tell it to inspect an ordinary object (typically from the Debugger or Listener), or by calling remote-inspect on the client side.

3.7.3.1 Accessing client side objects on the IDE side

Remote (client side) values can be used on the IDE side and the type of object affects how it is represented.

Remote numbers and characters are represented on the IDE side as their actual IDE side values.

Remote strings are represented on the IDE side as IDE side strings, which are copies of the string of the same element type. Note that, as a result, two separate occurrences in the IDE of the same client side string are not necessarily
the same object, and that modifying the characters in these strings does not affect the string on the client side.

Most other remote objects are represented in the IDE by remote handles (see below for exceptions). Handles are specific to a connection, so accessing the same remote object in the IDE multiple times over the same connection will always use the same (by \texttt{eq}) handle. However, accessing the same client object through different connections will use different handles, which are not equal at all, and there is no way to find if two handles from different connections refer to the same remote object.

Remote handles are printed like this:

\texttt{#<Remote ... >}

where the ... is the printing of the remote object by the client side.

Handles are opaque objects. The predicate \texttt{remote-object-p} can be used to check if an object is a remote object, and \texttt{remote-object-connection} returns the connection that the handle is associated with. If two handles are associated with the same connection, then they are \texttt{eq} if and only if they refer to the same object on the client side.

The generic function \texttt{get-inspector-values} has a method that specializes on handles to invoke \texttt{get-inspector-values} on the client side and return the results. Note that \texttt{get-inspector-values} also returns a setter, which allows you to set values inside the client’s object. This method makes the IDE Inspector and the CL functions \texttt{inspect} and \texttt{describe} work on remote handles.

Apart from the interface in the previous paragraphs, there is no useful way to access handles on the IDE side. However, you can access the underlying remote object by using \texttt{ide-eval-form-in-remote} or \texttt{ide-funcall-in-remote}, by sending a form containing the handle. For example, assuming the value of \texttt{my-remote-simple-vector} is a remote handle for a simple-vector, you can read its first element by:

\begin{verbatim}
(dg:ide-eval-form-in-remote `(svref ,my-remote-simple-vector 0))
\end{verbatim}

This will call \texttt{svref} on the client’s object that \texttt{my-remote-simple-vector} is a handle for, because the client side call receives the underlying object rather than the handle.
3 The Debugger

Each call to `ide-eval-form-in-remote` and `ide-funcall-in-remote` is associated with a specific connection, and only remote objects that are associated with the connection can be used in form arguments. Trying to use remote objects that are associated with another connection signals an error.

LispWorks represents certain client side conses/symbols as conses/symbols on the IDE side in cases where there is no need to access the remote object. For example, the lists that `get-inspector-values` returns are IDE side conses, and the symbols in the slot-names list are IDE side symbols (for symbols in packages that exist on the IDE side). By default, `ide-eval-form-in-remote` and `ide-funcall-in-remote` return handles to the values returned by the form, except for numbers, characters, strings and the top-level of lists. They have a keyword `:encoded-result` which gives you some control over whether the values are returned as handles or not.

When displaying the source code of a function, LispWorks uses source location information on the IDE side to find the source file. That means that the IDE side needs to have the same source files loaded as the client side. To find a subform inside the definition of a function, the debugger uses the information from the client side, which must be compiled with source-level-debugging (and kept if it is delivered) for this to work.

Invoking the Class Browser in the IDE for a remote object handle shows the class on the IDE side that has the same `class-name` as the class of the object on the client side. Calling `class-of` (and `type-of`) on the IDE side on a remove object handle return the internal class (and class name) of remote handles, which you should not be accessing.

### 3.7.3.2 Controlling the client side from the IDE side

The functions `ide-eval-form-in-remote` and `ide-funcall-in-remote` can be used to call functions on the client side (`ide-eval-form-in-remote` is used by the editor commands).

The function `ide-set-remote-symbol-value` can be used to set the global value of a symbol on the client side, which is a common operation. It is equivalent to calling `ide-funcall-in-remote` with `set`.

The function `ide-attach-remote-output-stream` can be used to create an output stream on the client side, such that any output into it will go to a
stream on the IDE side. It returns a remote object handle for the client side stream, which can then be used it calls to `ide-eval-form-in-remote` etc.

### 3.7.4 Troubleshooting
The sections below describe some unexpected problems that you might encounter when using remote debugging and suggest ways to solve them.

#### 3.7.4.1 Failing to open connections
There are some basic things to check first.

1. Make sure that you use the right pair of functions. Either:
   - `start-client-remote-debugging-server` on the client side and `ide-connect-remote-debugging` on the IDE side.
   - Or:

   ```lisp
   configure-remote-debugging-spec
   ```
   on the client side and `start-ide-remote-debugging-server` on the IDE side.

   When using `start-client-remote-debugging-server` and `ide-connect-remote-debugging`, `start-client-remote-debugging-server` must be called first.

   When using `configure-remote-debugging-spec` and `start-ide-remote-debugging-server`, `start-ide-remote-debugging-server` must be called before the connection is opened by the client. However, by default, `configure-remote-debugging-spec` delays opening the connection until it is needed so can be called first.

2. Check that you are either using the default port numbers, or you have changed them to the same number on both sides.

3. Check that you have the correct hostname in `configure-remote-debugging-spec` or `ide-connect-remote-debugging`.

4. On Android, you need to add the INTERNET permission to the application. The Android example has the line commented out in its Manifest file (`example-file "android/OthelloDemo/AndroidManifest.xml"`).
5. Check that the two machines can connect by TCP (for example, there is no firewall blocking connections).

6. Check that the functions you use return the expected values.

7. Specify the `:log-stream` argument for both sides, and check if anything is written to it.

If you cannot find the problem, then check that the connection works at the TCP level.

- If you run the client as the TCP server, evaluate the following on the client side:

```lisp
(setq *log-stream* <somewhere-that-you-can-see-it>)

(comm:start-up-server
 :function #'(lambda (socket)
           (format *log-stream* "Connected from ~a~%" (comm:get-socket-peer-address socket))
           (finish-output *log-stream*))
 :service dbg:*default-client-remote-debugging-server-port*)
```

and then on the IDE side evaluate:

```lisp
(comm:open-tcp-stream "<client-hostname>" dbg:*default-client-remote-debugging-server-port*)
```

The call to `open-tcp-stream` should return a stream, and the “Connected from” message should be printed to `*log-stream` on the client side.

- If you run the IDE as the TCP server, evaluate the following on the IDE side:

```lisp
(comm:start-up-server
 :function #'(lambda (socket)
           (format mp:*background-standard-output* "Connected from ~a~%" (comm:get-socket-peer-address socket))
           (finish-output mp:*background-standard-output*)
 :service dbg:*default-ide-remote-debugging-server-port*)
```

and then on the client side evaluate:

```lisp
(comm:open-tcp-stream "<ide-hostname>" dbg:*default-ide-remote-debugging-server-port*)
```
3.7 Remote debugging

The call to \texttt{open-tcp-stream} should return a stream, and the "Connected from" message should be printed to \texttt{Output} tab of the Listener or Editor in the IDE.

If you cannot connect as above then you need to fix the configuration of your machines to make it work. If you can connect as above, but the remote debugging does not connect after you did all the checks then contact LispWorks support for help.

3.7.4.2 The Inspector does not show slots in a remote object

The most likely explanation for this is that the underlying connection was closed. Check if the remote object prints with "\{closed connection\}".

3.7.5 Advanced usage - multiple connections

3.7.5.1 Client side connection management

By default, the connection opened by the client side functions is reused whenever a connection is needed. This is normally all that is required, but sometimes it is useful to have a better control.

The client has a default connection (a global value), and a switch that enables using the default connection (the enabling switch). Both \texttt{configure-remote-debugging-spec} and \texttt{start-client-remote-debugging-server} have keyword arguments \texttt{:setup-default} and \texttt{:enable}, which control setting the default connection, and whether to enable using it. The default for both is true. The default connection and switch are used by the APIs that need a connection (mainly when the debugger is entered, but also \texttt{remote-inspect} and \texttt{start-remote-listener}), so by default the connection that was opened last is used.

The value of the enabling switch can be set globally by \texttt{set-remote-debugging-connection} or in a dynamic extent by \texttt{with-remote-debugging-connection}. These functions also allow you to specify a specific connection to use, rather than the default. Note that \texttt{with-remote-debugging-connection} and \texttt{set-remote-debugging-connection} do not affect the default connection, only the enabling switch.
The function `set-default-remote-debugging-connection` can be used to set the default connection.

`start-client-remote-debugging-server` and `configure-remote-debugging-spec` take also a keyword `:open-callback`, which specifies a callback that is called whenever a connection is created. You can use this callback to store the connection somewhere for later use, for example in a call to `set-default-remote-debugging-connection`, `set-remote-debugging-connection` or `with-remote-debugging-connection`.

The value of the `:setup-default` keyword to `configure-remote-debugging-spec` can also be `:delayed`, which means that the connection is not opened immediately, but when it is opened, it is set as the default.

When the debugger is entered, it first checks the value of the enabling switch. If it is set to a connection then that connection is used. If it is set to `t`, then the default connection is used. Otherwise, it checks if a host was configured (globally by `configure-remote-debugging-spec` or in a dynamic extent by `with-remote-debugging-spec`) and tries to open a connection to it. If that succeeds, it decides according to the `:setup-default` and `:enable` arguments whether to set the default connection and the enabling switch, and then use this connection. If `:setup-default` is `nil`, then the connection is closed when the Remote Debugger is closed.

When a connection is closed, all remote object handles in the IDE that were created using it become invalid. For example, if you use `configure-remote-debugging-spec` with `:setup-default nil`, and later the Remote Debugger was raised and you inspected an argument of a function and then closed the Remote Debugger, then the Inspector will fail to access slots in this object. For this reason, it is usually better to have a permanently open default connection, so the Inspector can be still be used after the Remote Debugger has been closed. A single connection is also more efficient, but the effect of this is small.

`start-remote-listener` behaves the same as the debugger when it tries to find a connection.

`remote-inspect` first checks if it can use an existing connection the same way as the debugger. However, if it cannot, it tries to use a connection used previously by an Inspector. If this does not work, it tries the default connection (if any) even if the enabling switch is `nil`, and if all these fail, it opens a connec-
tion using the configured host if any, and remembers it for the next time (unless configured to set this connection as the default). The Inspector behaves differently to the Debugger/Listener is because there is no obvious place where a temporary connection should be closed in the case of the Inspector, compared to the Debugger and Listener where closing the GUI tool is the natural place to close the connection if it is temporary.

In the simplest usage, you will have one connection that is used for everything. The next level of complexity is to have one connection, but control dynamically whether to use it or not, either globally by calls to `set-remote-debugging-connection`, or in a dynamic extent using `with-remote-debugging-connection`.

For more complex usage, you can use the `open-callback` to record the connections that you have opened, and then use them in `set-default-remote-debugging-connection` or `set-remote-debugging-connection` or `with-remote-debugging-connection` to tell the debugger/listener/inspector which connection to use.

The functions described in “Common (both IDE and client) connection functions” on page 36 can also be used on the client side.

### 3.7.5.2 IDE side connection management

Normally you do not need to manage remote debugging connections on the IDE side, but sometimes it may be useful.

`ide-list-remote-debugging-connections` returns a list of connections.

`ide-find-remote-debugging-connection` can be used to find a connection. This is used by default by the IDE side functions that need a connection (`ide-eval-form-in-remote`, `ide-funcall-in-remote`, `ide-set-remote-symbol-value` and `ide-attach-remote-output-stream`), and the Editor commands.

`ide-set-default-remote-debugging-connection` can be used to set the default connection, which is what the Editor commands use, and affects what `ide-find-remote-debugging-connection` returns.

You can get the connection from a remote object handle by using `remote-object-connection`. 
The functions described in “Common (both IDE and client) connection functions” on page 36 can also be used on the IDE side.

### 3.7.5.3 Common (both IDE and client) connection functions

You can close a connection by using `close-remote-debugging-connection`. Closing a connection causes the other side to be closed too. Closing the default connection causes the default to be set to `nil`.

You can use `ensure-remote-debugging-connection` to check if a connection is alive. `ensure-remote-debugging-connection` can be called with any object and returns `nil` for any non-connection object or a connection object that is closed.

You can use `remote-debugging-connection-add-close-cleanup` to add a "cleanup", which is a callback function that is called when the connection is closed, and `remote-debugging-connection-remove-close-cleanup` to remove a previously added cleanup.

The function `remote-debugging-connection-peer-address` can be used to check what machine is on the other side.

The function `remote-debugging-connection-name` can be used to find the name of a connection.

Each opened connection has a dedicated Lisp process that handles communications through it, which you can see by listing processes using `(mp:ps)` or in the Process Browser tool. The name of the connection appears in the name of the process. You can forcibly close a connection by using `process-terminate` on the process or from the Process Browser.
LispWorks provides two inspectors. One is for use with the LispWorks IDE, and is described in the *LispWorks IDE User Guide*. The other is the REPL inspector, which uses a stream interface, and can be used on any terminal (in particular within the LispWorks IDE Listener tool). Both inspectors allow you to traverse complex data structures interactively and to destructively modify components of these structures. However, the two inspectors are quite different. No attempt has been made to make their usage compatible and instead each inspector is designed to exploit to the full the particular environment facilities available.

The REPL inspector provides a simple inspector facility which can be used on a stream providing line breaks as the only type of formatting. It is built on top of the describe function which is briefly described below and modifies the top level loop in a similar way to the debugger (see Chapter 3, “The Debugger”).

### 4.1 Describe

The function describe displays the slots of composite data structures in a manner dependent on the type of the object. The slots are labeled with a name where appropriate, or otherwise with a number.

The example below shows the result of calling describe on a simple list.
USER 7 > (setq countries '("Chile" "Peru" "Paraguay" "Brazil"))
("Chile" "Peru" "Paraguay" "Brazil")

USER 8 > (describe countries)
("Chile" "Peru" "Paraguay" "Brazil") is a CONS
[0] : "Chile"
[1] : "Peru"
[3] : "Brazil"

describe describes slots recursively up to a limit set by the special variable *describe-level*. Note that only arrays, structures and conses are printed recursively. The slots of all other object types are only printed when at the top level of describe.

*describe-level* has an initial value of 1.

The symbols *describe-print-level* and *describe-print-length* are similar in effect to *trace-print-level* and *trace-print-length*. They control, respectively, the depth to which nested objects are printed (initial value 10), and the number of components of an object which are printed (initial value 10).

To customize describe, define new methods on the generic function describe-object.

### 4.2 Inspect

The function inspect is an interactive version of describe. It displays objects in a similar way to describe. Entering the inspector causes a new level of the top loop to be entered with a special prompt indicating that the inspector has been entered and showing the current inspector level.

In the modified top loop, if you enter a slot name, that slot is inspected and the current object is pushed onto an internal stack of previously inspected objects. The special variables $, $$, and $$$ are bound to the top three objects on the inspector stack.
The following keywords are treated specially as commands by the inspector.

Table 4.1 Inspector commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>:cv</td>
<td>Display current values of control variables.</td>
</tr>
<tr>
<td>:d</td>
<td>Display current object.</td>
</tr>
<tr>
<td>:dm</td>
<td>Display more of current object.</td>
</tr>
<tr>
<td>:dr</td>
<td>Display rest of current object.</td>
</tr>
<tr>
<td>:h</td>
<td>Display help on inspector commands.</td>
</tr>
<tr>
<td>:im</td>
<td>Recursively invoke a new inspector. (m) is an object to inspect.</td>
</tr>
<tr>
<td>:m</td>
<td>Change the inspection mode — see Section 4.3 on page 40.</td>
</tr>
<tr>
<td>:q</td>
<td>Quit current inspector.</td>
</tr>
<tr>
<td>:s n v</td>
<td>Sets slot (n) to value (v).</td>
</tr>
<tr>
<td>:sh</td>
<td>Show inspector stack.</td>
</tr>
<tr>
<td>:u int</td>
<td>Undo last inspection. If you supply an optional integer argument, (int), then the last (int) inspections are undone.</td>
</tr>
<tr>
<td>:ud</td>
<td>Undo last inspection and redisplay current object.</td>
</tr>
</tbody>
</table>

You can get brief help listing these commands by entering :? at the inspector prompt.

The control variables \texttt{*inspect-print-level*} and \texttt{*inspect-print-length*} are similar to \texttt{*describe-print-level*} and \texttt{*describe-print-length*} (see above).

\texttt{:dm} displays more slots of the current object. If the object has more than \texttt{*describe-length*} slots, then the first \texttt{*describe-length*} will be printed, followed by an ellipsis and then

\( (:dm\ or\ :dr\ for\ more) \)

If you enter the command \texttt{:dm} at the prompt it displays the next \texttt{*describe-length*} slots, and if you enter \texttt{:dr} it displays all the remaining slots. This
only works on the last inspected object, so if you recursively inspect a slot and come back, :dm does not do anything useful. Typing :d lets you view the object again.

:ud is equivalent to typing :u followed by :d.

### 4.3 Inspection modes

The :m command displays and changes the current inspection mode for an inspected value. The session below demonstrates how it works:

```
CL-USER 128 > (inspect "a
string with
newlines in it")
"a
string with
newlines in it" is a SIMPLE-BASE-STRING
0   #\a
1   #\Newline
2   #\s
3   #\t
4   #\r
5   #\i
6   #\n
7   #\g
8   #\Space
9   #\w
10  #\i
11  #\t
12  #\h
13  #\Newline
14  #\n
15  #\e
16  #\w
17  #\l
18  #\i
19  #\n ........ (:dm or :dr for more)
```

```
CL-USER 129 : Inspect 1 > :m
* 1. SIMPLE-STRING
  2. LINES
```

The :m produces an enumerated list of inspection modes for this value.
4.3 Inspection modes

The asterisk next to

* 1. SIMPLE-STRING

means that SIMPLE-STRING is the current inspection mode.

You can change mode by typing :m followed by the name or number of another mode. To change to LINES mode:

```
CL-USER 130 : Inspect 1 > :m 2
"a
string with
newlines in it" is a SIMPLE-BASE-STRING
0    a
1    string with
2    newlines in it
```

4.3.1 Hash table inspection modes

There are five hash table inspection modes. They can be accessed in either the LispWorks IDE Inspector tool or the REPL inspector.

A brief introduction to the representation of hash tables is necessary so that you can fully understand what you gain from the new modes.

Internally, a hash table is a structure containing, among other things,

- a big vector
- size and growth information
- accessing functions.

When keys and values are added to the table, sufficiently similar keys are converted into the same index in the vector. When this happens, the similar keys and values are kept together in a chain that hangs off this place in the vector.

The different inspection modes provide views of different pieces of this structure:

**HASH-TABLE** This mode is the “normal” view of a hash table; as a table of keys and values. When you inspect an item you inspect the value of the item.
STRUCTURE  This mode provides a raw view of the whole hash table structure. When you inspect an item you are inspecting the value of that slot in the hash table structure.

ENUMERATED-HASH-TABLE

This mode is a variation of the normal view, where a hash table is viewed simply as a list of lists. When you inspect an item you are inspecting a list containing a key and a value.

HASH-TABLE-STATISTICS

This mode shows how long the chains in the hash table are, so that you can tell how efficiently it is being used. For example, if all chains contained fewer than two items the hash table would be being used well.

HASH-TABLE-HISTOGRAM

This mode shows the statistical information from HASH-TABLE-STATISTICS as a histogram.

Here is an example of hash table inspection.
4.3 Inspection modes

CL-USER 1 > (defvar *hash* (make-hash-table))

CL-USER 2 > (setf (gethash 'lisp *hash*) 'programming
               (gethash 'java *hash*) 'programming
               (gethash 'c *hash*) 'programming
               (gethash 'c++ *hash*) 'programming
               (gethash 'english *hash*) 'natural
               (gethash 'german *hash*) 'natural)

CL-USER 3 > (inspect *hash*)

#<EQL Hash Table{6} 21C15D97> is a HASH-TABLE
C++          PROGRAMMING
JAVA         PROGRAMMING
ENGLISH      NATURAL
C            PROGRAMMING
GERMAN       NATURAL
LISP         PROGRAMMING

CL-USER 4 : Inspect 1 > :m
* 1. HASH-TABLE
  2. STRUCTURE
  3. ENUMERATED-HASH-TABLE
  4. HASH-TABLE-STATISTICS
  5. HASH-TABLE-HISTOGRAM

STRUCTURE mode displays the raw representation of the hash table:
In **ENUMERATED-HASH-TABLE** mode you can recursively inspect keys and values by entering the index. This is especially useful in cases where the key or value is unreadable and so cannot be entered into the REPL:

```lisp
CL-USER 6 : Inspect 1 > :m 3

#:EQL Hash Table{6} 21C15D97> is an Enumerated HASH TABLE
0  (C++ PROGRAMMING)
1  (JAVA PROGRAMMING)
2  (ENGLISH NATURAL)
3  (C PROGRAMMING)
4  (GERMAN NATURAL)
5  (LISP PROGRAMMING)

CL-USER 7 : Inspect 1 > 5

(LISP PROGRAMMING) is a LIST
0  LISP
1  PROGRAMMING

CL-USER 8 : Inspect 2 > :u
```
The **HASH-TABLE-STATISTICS** mode shows that *hash* has 31 chains, of which 25 are empty and 6 have one entry:

```
CL-USER 9 : Inspect 1 > :m 4
#<EQL Hash Table{6} 21C15D97> is a HASH-TABLE (statistical view)
chain of length 0 : 31
chain of length 1 : 6
```

In **HASH-TABLE-HISTOGRAM** mode the same information is represented as a histogram:

```
CL-USER 10 : Inspect 1 > :m 5
#<EQL Hash Table{6} 21C15D97> is a HASH-TABLE (histogram view)
chain of length 0 : ************
chain of length 1 : *
```

```
CL-USER 11 : Inspect 1 > :q
#<EQL Hash Table{6} 21C15D97>
```
4 The REPL Inspector
The trace facility is a debugging aid enabling you to follow the execution of particular functions. At any time there are a set of functions (and macros and methods) which are being monitored in this way. The normal behavior when a call is made to one of these functions is for the function’s name, arguments and results to be printed out by the system. More generally you can specify that particular forms should be executed before or after entering a function, or that certain calls to the function should cause it to enter the main debugger. Tracing of a function continues even if the function is redefined.

The standard way of arranging for functions to be traced is to call the macro \texttt{trace} with the symbols of the functions (or macros or generic functions) concerned. In addition it is possible to restrict tracing to a particular method (rather than a generic function) as described in “Tracing methods” on page 57. The trace facility also handles recursive and nested calls to the functions concerned.

5.1 Simple tracing

This section shows you how to perform simple traces.

1. Enter this definition of the factorial function \texttt{fac} into the listener:
(defun fac (n)
  (if (= n 1) 1
      (* n (fac (- n 1)))))

2. Now trace the function by entering the following into the listener.

   (trace fac)

3. Call the function fac as follows:

   (fac 3)

The following trace output appears in the listener.

   0 FAC > ...
     >> N : 3
   1 FAC > ...
     >> N : 2
     2 FAC > ...
       >> N : 1
     2 FAC < ...
       << VALUE-0 : 1
   1 FAC < ...
     << VALUE-0 : 2
   0 FAC < ...
     << VALUE-0 : 6

Upon entry to each traced function call, trace prints the following information:

- The level of tracing, that is, the number of recursive entries to trace.
- The traced function name.
- The arguments and their values for the current call.

Each line is indented according to the level of tracing for the call.

> denotes entry to a function, and >> denotes an argument.

Upon exit from each traced function call, trace prints the following information:

- The level of tracing.
- The traced function name.
- The returned values for the current call.
< denotes exit from a function, and << denotes a returned value.
Output produced in this way is always sent to a special stream, *trace-output*, which is either associated with the listener, or with background output.

Calling trace with no arguments produces a list of all the functions currently being traced. In order to cease tracing a function the macro untrace should be called with the function name. All tracing can be removed by calling untrace with no arguments.

```
CL-USER 5 > (untrace fac)
(FAC)
CL-USER 6 > (fac 4)
  24
```

### 5.2 Tracing options

There are a number of options available when using the trace facilities, which allow you both to restrict or expand upon the information printed during a trace. For instance, you can restrict tracing of a function to a particular process, or specify additional actions to be taken on function call entry and exit.

Note that the options and values available only apply to a particular traced function. Each traced function has its own, independent, set of options.

This section describes the options that are available. Each option can be set as described in the next subsection.

#### 5.2.1 Evaluating forms on entry to and exit from a traced function

**:before Trace keyword**

**:before list-of-forms**

If non-nil, the list of forms is evaluated on entry to the function being traced. The forms are evaluated and the results printed after the arguments to the function.
Here is an example of its use. *traced-arglist* is bound to the list of arguments given to the function being traced. In this example, it is used to accumulate a list of all the arguments to fac across all iterations.

1. In the listener, initialize the variable args-in-reverse as follows:
   
   ```lisp
   (setq args-in-reverse ())
   ```

2. For the fac function used earlier, set the value of :before as follows:
   
   ```lisp
   (trace (fac :before ((push (car *traced-arglist*) args-in-reverse)))
   ```

3. In the listener, evaluate the following form:
   
   ```lisp
   (fac 3)
   ```

After evaluating this form, args-in-reverse has the value (1 2 3), that is, it lists the arguments which fac was called with, in reverse order.

 trace keyword

:after

:after list-of-forms

If non-nil, this option evaluates a list of forms upon return from the function to be traced. The forms are evaluated and the results printed after the results of a call to the function.

This option is used in exactly the same way as :before. For instance, using the example for :before as a basis, create a list called results-in-reverse, and set the value of :after so that (car *traced-results*) is pushed onto this list. After calling fac, results-in-reverse contains the results returned from fac, in reverse order.

Note also that *traced-arglist* is still bound.

5.2.2 Evaluating forms without printing results

 trace keyword

:eval-before

:eval-before list-of-forms
This option allows you to supply a list of forms for evaluation upon entering the traced function. The forms are evaluated after printing out the arguments to the function, but unlike :before their results are not printed.

.eval-after

 eval-after list-of-forms

This option allows you to supply a list of forms for evaluation upon leaving the traced function. The forms are evaluated after printing out the results of the function call, but unlike :after their results are not printed.

5.2.3 Using the debugger when tracing

:break

 break form

If form evaluates to non-nil, the debugger is entered directly from trace. If it returns nil, tracing continues as normal. This option lets you force entry to the debugger by supplying a form as simple as t.

Upon entry to the traced function, the standard trace information is printed, any supplied :before forms are executed, and then form is evaluated.

:break-on-exit

 break-on-exit form

Like :break, this option allows you to enter the debugger from trace. It differs in that the debugger is entered after the function call is complete.

Upon exit from the traced function, the standard trace information is printed, and then form is evaluated. Finally, any supplied :after forms are executed.
5. The Trace Facility

:backtrace  Trace keyword

:backtrace backtrace

Generates a backtrace on each call to the traced function. backtrace can be
any of the following values:

:quick       Like the :q debugger command.
t           Like the :b debugger command.
:verbose     Like the :b :verbose debugger command.
:bug-form    Like the :bug-form debugger command.

5.2.4 Entering stepping mode

:step  Trace keyword

:step form

When non-nil, this option puts the trace facility into stepper mode,
where interpreted code is printed one step of execution at a time.

5.2.5 Configuring function entry and exit information

:entrycond  Trace keyword

:entrycond form

This option controls the printing of information on entry to a traced
function. form is evaluated upon entry to the function, and information
is printed if and only if form evaluates to t. This allows you to turn off
printing of function entry information by supplying a form of nil, as in
the example below.

:exitcond  Trace keyword

:exitcond form

This option controls the printing of information on exit from a traced
function. form is evaluated upon exit from the function, and, like
5.2 Tracing options

:entrycond, information is printed if and only if form evaluates to a non-nil value. This allows you to turn off printing of function exit information by supplying a form of nil.

An example of using :exitcond and :entrycond is shown below:

1. For the fac function, set the values of :entrycond and :exitcond as follows.

   (trace (fac :entrycond (evenp (car *traced-arglist*)))
            :exitcond (oddp (car *traced-arglist*))))

   Information is only printed on entry to fac if the argument passed to fac is even. Conversely, information is only printed on exit from fac if the argument passed to fac is odd.

2. Enter the following call to <Code>fac</Code> in a listener:

   CL-USER 24 > (fac 10)

   The tracing information printed is as follows:

   0 FAC > ...
     >> N : 10
   2 FAC > ...
     >> N : 8
   4 FAC > ...
     >> N : 6
   6 FAC > ...
     >> N : 4
   8 FAC > ...
     >> N : 2
   9 FAC < ...
      << VALUE-0 : 1
   7 FAC < ...
      << VALUE-0 : 6
   5 FAC < ...
      << VALUE-0 : 120
   3 FAC < ...
      << VALUE-0 : 5040
   1 FAC < ...
      << VALUE-0 : 362880
5.2.6 Directing trace output

:**trace-output**

This option allows you to direct trace output to a stream other than the listener in which the original function call was made. By using this you can arrange to dispatch traced output from different functions to different places.

Consider the following example:

1. In the listener, create a file stream as follows:

   ```lisp
   CL-USER 1 > (setq str (open "trace.txt" :direction :output))
   Warning: Setting unbound variable STR
   #<STREAM::LATIN-1-FILE-STREAM C:\temp\trace.txt>
   ``

2. Set the value of the **:trace-output** option for the function **fac** to **str**.

3. Call the **fac** function, and then close the file stream as follows:

   ```lisp
   CL-USER 138 > (fac 8)
   40320
   CL-USER 139 > (close str)
   T
   ``

   Inspect the file **trace.txt** in order to see the trace output for the call of **(fac 8)**.

5.2.7 Restricting tracing

:**process**

This lets you restrict tracing of a function to a particular process. If **process** evaluates to **t**, then the function is traced from within all processes (this is the default). Otherwise, the function is only traced from within the process that **process** evaluates to.
5.2 Tracing options

:when

This lets you invoke the tracing facilities on a traced function selectively. Before each call to the function, \textit{form} is evaluated. If \textit{form} evaluates to \texttt{nil}, no tracing is done. The contents of \texttt{*traced-arglist*} can be examined by \textit{form} to find the arguments given to \texttt{trace}.

5.2.8 Storing the memory allocation made during a function call

:allocation

If \textit{form} is non-nil, this prints the memory allocation, in bytes, made during a function call. The symbol that \textit{form} evaluates to is used to accumulate the amount of memory allocated between entering and exiting the traced function.

Note that this symbol continues to be used as an accumulator on subsequent calls to the traced function; the value is compounded, rather than over-written.

Consider the example below:

1. For the \texttt{fac} function, set the value of \texttt{:allocation} to \texttt{$$fac-alloc}.
2. In the listener, call \texttt{fac}, and then evaluate \texttt{$$fac-alloc}.

\begin{verbatim}
CL-USER 152 > $$fac-alloc
744
\end{verbatim}

5.2.9 Tracing functions from inside other functions

:inside

The functions given in the argument to \texttt{:inside} should reference the traced function in their implementation. The traced function is then only
5. The Trace Facility

traced in calls to any function in the list of functions, rather than in direct
calls to itself.

For example:

1. Define the function `fac2`, which calls `fac`, as follows:

   ```lisp
   (defun fac2 (x)
      (fac x))
   ```

2. For the `fac` function, set the value of `:inside` to `fac2`:

   ```lisp
   (trace (fac :inside fac2))
   ```

3. Call `fac`, and notice that no tracing information is produced.

   ```lisp
   CL-USER 2 > (fac 3)
   6
   ```

4. Call `fac2`, and notice the tracing information.

   Evaluate `(fac2 3)`, and notice the tracing information.

   ```lisp
   0 FAC > ...
   >> N : 3
   1 FAC > ...
   >> N : 2
   2 FAC > ...
   >> N : 1
   2 FAC < ...
   << VALUE-0 : 1
   1 FAC < ...
   << VALUE-0 : 2
   0 FAC < ...
   << VALUE-0 : 6
   ```

5.3 Example

The following example illustrates how `trace` may be used as a debugging
tool. Suppose that you have defined a function `f`, and intend its first argument to be a non-negative number. You can trap calls to `f` where this is not true, providing an entry into the main debugger in these cases. It is then possible for you to investigate how the problem arose.

To do this, you specify a `:break` option for `f` using `trace`. If the form following this option evaluates to a non-nil value upon calling the function, then the
debugger is entered. In order to inspect the first argument to the function \( f \), you have access to the variable \(*\text{traced-arglist}*\). This variable is bound to a list of the arguments with which the function was called, so the first member of the list corresponds to the first argument of \( f \) when tracing \( f \).

```
CL-USER 1 > (defun f (a1 a2) (+ (sqrt a1) a2))
F
CL-USER 2 > (trace (f :break (< (car *traced-arglist*) 0)))
(F)
CL-USER 3 > (f 9.0 3)
0 F > ...
  >> A1 : 9.0
  >> A2 : 3
0 F < ...
<< VALUE-0 : 6.0
6.0
CL-USER 4 > (f -16.0 3)
0 F > ...
  >> A1 : -16.0
  >> A2 : 3
```

Break on entry to \( F \) with \(*\text{TRACED-ARGLIST}*\) (-16.0 3).
1 (continue) Return from break.
2 Continue with trace removed.
3 Continue traced with break removed.
4 Continue and break when this function returns.
5 (abort) Return to level 0.
6 Return to top loop level 0.

Type :b for backtrace or :c <option number> to proceed.
Type :bug-form "<subject>" for a bug report template or :? for other options.

```
CL-USER 5 : 1 >
```

### 5.4 Tracing methods

You can also trace methods (primary and auxiliary) within a generic function. The following example shows how to specify any qualifiers and specializers.

1. Type the following methods into the listener:
(defmethod foo (x)
  (print ‘there))

(defmethod foo :before ((x integer))
  (print ‘hello))

2. Next, trace only the second of these methods by typing the following definition spec.

   (trace (method foo :before (integer)))

3. Test that the trace has worked by calling the methods in the listener:

   CL-USER 226 > (foo ’x)
   THERE
   THERE

   CL-USER 227 > (foo 4)
   0 (METHOD FOO :BEFORE (INTEGER)) > (4)
   HELLO
   0 (METHOD FOO :BEFORE (INTEGER)) < (HELLO)
   THERE
   THERE

   CL-USER 228 >

5.5 Tracing subfunctions

Subfunctions are functions that are defined inside the body of other functions rather than by top level definers like defun, defmethod, etc. To trace such a subfunction, call trace with a "subfunction dspec" in one of these forms:

- (subfunction sub-name parent-dspec)
- (flet sub-name parent-dspec)
- (labels sub-name parent-dspec)

See “Subfunction dspecs” on page 85 for details. All of the keywords that trace takes have the same effect for subfunction tracing.

The behavior when tracing a subfunction is somewhat different from tracing other function.
5.6 Trace variables

- `trace` modifies the parent of the subfunction such that future execution of the code that creates the subfunction will create it traced.

- A subsequent call to `trace` with the same dspec re-modifies the parent, and also causes any of the subfunctions that were already created traced to change their tracing behavior (as defined by the keywords to `trace`) to the behavior specified by the latest call to `trace`.

- `untrace` with the same dspec returns the parent function to its original state and switches off tracing for the subfunctions that were created traced.

- A call to `trace` with the same dspec after `untrace` modifies the parent as in the first point above, but has no effect on any subfunction that was created by the parent before the call to `untrace`.

Tracing of subfunction works only for compiled functions.

5.5.1 Notes on subfunction names

Anonymous lambdas are named by the compiler using integers.

You can find the dspec of a given subfunction by calling `object-dspec` on the subfunction. You can also construct it from the printed representation of the subfunction, which contains the sub-name and the parent-dspec.

A subfunction can be given an name using a `hcl:lambda-name` declaration (see `declare`, page 651). If this is of the form `(subfunction sub-name)`, then the dspec of the subfunction will contain both `sub-name` and the correct parent-dspec. However, if it has any other form, then dspec will be that name and you will need to know the parent-dspec in order to construct the subfunction dspec.

5.6 Trace variables

- `*max-trace-indent*`  
  The maximum indentation used during output from `trace`.

- `*trace-indent-width*`
The additional amount by which tracing output is indented upon entering a deeper level of nesting.

*trace-level*

The current depth of tracing.

cl:*trace-output*

The stream to which tracing sends its output by default.

*traced-arglist*

The variable that holds the arguments given to the traced function.

*traced-results*

The variable that holds the results from the traced function.

The following four variables allow you to control the style of tracing output separately from normal printing:

*trace-print-circle*

The value to which *print-circle* is bound during output from trace.

*trace-print-length*

The value to which *print-length* is bound during output from trace.

*trace-print-level*

The value to which *print-level* is bound during output from trace.

*trace-print-pretty*

The value to which *print-pretty* is bound during output from trace.

5.7 Troubleshooting tracing

This section describes some of the common problems seen when tracing, with suggestions to overcome these.
5.7.1 Excessive output

In general it is not useful to trace `cl:length` and other base-level functions unconditionally because they are called too frequently by LispWorks itself. It may be useful to trace these functions in a limited fashion, using the `trace` options `:inside` or `:when`.

5.7.2 Missing output

There are two common reasons for not seeing calls you expect in trace output.

5.7.2.1 Compiled code may not call the functions you expect

There are many other optimizations built-in to the LispWorks compiler, which affect code generated according to the compiler qualities in effect at compile-time. For example if the compiler was set to inline structure accessors, then tracing structure accessors in code compiled with that setting will produce no output.

While debugging, you could re-compile the code at higher safety or run it interpreted, to obtain the trace output.

5.7.2.2 trace works on function names, not function objects

`trace` works by tracing function names, not function objects. Therefore tracing function objects, for example by

```lisp
(trace #'foo)
```

will not yield any trace output. Instead you need to do

```lisp
(trace foo)
```

Also, if the symbol `foo` is traced, then code which invokes `foo` by

```lisp
(funcall (symbol-function 'foo) ...)
```

or equivalently

```lisp
(funcall #'foo ...)
```

will not produce any trace output.
The correct approach is to use \( \text{funcall } '\text{foo} \ldots \) instead of \( \text{funcall } #'\text{foo} \ldots \).
The Advice Facility

The advice facility provides a mechanism for altering the behavior of existing functions. As a simple application of this, you may supplement the original function definition by supplying additional actions to be performed before or after the function is called. Alternatively, you may replace the function with a new piece of code that has access to the original definition, but which is free to ignore it altogether and to process the arguments to the function and return the results from the function in any way you decide. The advice facility allows you to alter the behavior of functions in a very flexible manner, and may be used to engineer anything from a minor addition of a message, to a major modification of the interface to a function, to a complete change in the behavior of a function. This facility can be helpful when debugging, or when experimenting with new versions of functions, or when you wish to locally change some functionality without affecting the original definition.

Note: It can be dangerous to put advice on system functions or functions used at low-level by the system. In general, advising a basic Common Lisp function (that is, a simple function for manipulating simple objects such as `reverse`) is dangerous, because the implementation may use it.
6.1 Defining advice

Each change that is required should be specified using the `defadvice` macro. This defines a new body of code to be used when the function is called; this piece of code is called a piece of advice. Consider the following example:

```lisp
(defadvice
capi:prompt-for-file pff-1 :before
  (message &key &allow-other-keys)
  (format t "~&Prompting for file with message ~S~%" message))
```

Here `defadvice` is given the name of the function you want to alter, a name for the piece of advice, and the keyword :before to indicate that you want the code carried out before `capi:prompt-for-file` is called. The rest of the call to `defadvice` specifies the additional behavior required, and consists of the lambda list for the new piece of advice and its body (the lambda list may specify keyword parameters and so forth). The advice facility arranges that `pff-1` is invoked whenever `capi:prompt-for-file` is called, and that it receives the arguments to `capi:prompt-for-file`, and that directly after this the original definition of `capi:prompt-for-file` is called.

After executing this advice definition, demonstrate it by selecting the menu command `File > Open` in the LispWorks IDE. The message appears in the Output tab.

Pieces of advice may be given to be executed after the call by specifying :after instead of :before in the call to `defadvice`. So if you wished to add further code to be performed after `capi:prompt-for-file` you could also define:

```lisp
(defadvice
capi:prompt-for-file pff-2 :after
  (message &rest args)
  (format t
    "~&The other arguments to prompt-for-file were: -S-%" args))
```

Note that `pff-2` also receives the arguments to `capi:prompt-for-file`, which are reported by the body.

Note also that `defadvice` works on function names, not function objects, like `trace`. See “trace works on function names, not function objects” on page 61 for details.
6.2 Combining the advice

We have already seen how a before and an after piece of advice may be combined, and this section describes the general algorithm. There are three types of advice: before, after and around. These resemble before, after and around methods in CLOS. There may be several pieces of each type of advice present for a particular function.

The first step in working out how the combination is done is to order the pieces of advice. All the around advice comes first, then all the before advice, then the original definition, and lastly the after advice. The order within each of the around, before and after sections defaults to the order in which the pieces of advice were defined (that is most recent first). See defadvice, page 1100 for details of how to control the ordering of advice within each section.

The remainder of this section discusses what happens when a function that has advice is called.

6.2.1 :before and :after advice

First we deal with the case when there is no around advice present. Here each of the pieces of before advice are called in turn, with the same arguments that were given to the function, next the original definition is called with these arguments, and finally each of the pieces of after advice is called in reverse order with the same arguments (so that by default the most recently added piece of after advice is invoked last). The results returned by the function call are the values produced by the last piece of after advice to be called (if there is one), or by the original definition (if there is no after advice).

Note that none of these bits of code should destructively modify the arguments that they receive. Adding a piece of before advice thus provides a simple way of specifying some additional action to be performed before the original definition, and before any older bits of before advice. Adding a piece of after advice allows you to specify extra actions to be performed after the original definition, and after any older bits of after advice. The advice facility automatically links together these bits of advice with the original function definition.
6.2.2 :around advice

Next we shall discuss the use of around advice, which provides you with greater control than do before and after advice. Let us suppose that a function that has some around advice is called. The arguments to the function are passed to the code associated with the first piece of around advice in the ordering, and the values returned by that piece of advice are the results of the function. There is no requirement for the advice to invoke any other pieces of advice, nor to call the original definition of the function.

However the code for any piece of around advice has access to the next member of the ordering, which it may invoke any number of times by calling \texttt{call-next-advice}. So it is possible for each piece of around advice to call its successor in the ordering if this is desired, and then the bits of around advice are called in turn in a similar fashion to our earlier description for before and after advice. However in the case of around advice the decision whether or not to call the next piece of advice is directly under your control, and you are free to modify the arguments received by the piece of advice, and to choose the arguments to be given to the next piece of advice if it is called.

If the last piece of around advice in the ordering calls \texttt{call-next-advice}, then it invokes the combination of before and after advice and the original definition that was discussed earlier. That is, the arguments to the call are given in the sequence described above to each of the before pieces of advice, then to the original definition and then to the after pieces of advice. The call to \texttt{call-next-advice} returns with the values produced by the last of these subsidiary calls, and the around advice may use these values in any way.

6.3 Removing advice

The macro \texttt{delete-advice} (or the function \texttt{remove-advice}) may be used to remove a named piece of advice. Since several pieces of advice may be attached to a single functional definition, the name must be supplied to indicate which one is to be removed.

\begin{verbatim}
CL-USER 40 > (delete-advice capi:prompt-for-file pff-1) NIL

CL-USER 41 > (delete-advice capi:prompt-for-file pff-2) NIL
\end{verbatim}
6.4 Advice for macros and methods

As well as attaching advice to ordinary functions, it may also be attached to macros and methods.

In the case of a macro, advice is linked to the macro’s expansion function, and so any before or after advice receives a copy of the arguments given to this expansion function (normally the macro call form and an environment). A simple example:

```lisp
CL-USER 45 > (defmacro twice (b) `(+ ,b ,b))
TWICE

CL-USER 46 > (defadvice (twice before-twice :before) (call-form env) (format t "~%Twice with environment ~A and call-form ~A" env call-form))
NIL

CL-USER 47 > (twice 3)
Twice with environment NIL and call-form (TWICE 3)
6
```

Note that the advice is invoked when the macro’s expansion function is used. So if the macro is present within a function that is being compiled, then the advice is invoked during compilation of that function (and not when that function is finally used).

In the case of a method, the call to `defadvice` must also specify precisely to which method the advice belongs. A generic function may have several methods, so the call to `defadvice` includes a list of classes. This must correspond exactly to the parameter specializers of one of the methods for that generic function, and it is to that method that the advice is attached. For example:
CL-USER 45 > (progn
    (defclass animal ()
      (genus habitat description
      (food-type :accessor eats)
      (happiness :accessor how-happy)
      (eaten :accessor eaten :initform nil)))
    (defclass cat (animal)
      ((food-type :initform 'fish)))
    (defclass elephant (animal)
      (memory (food-type :initform 'hay)))
    (defmethod feed ((animal animal))
      (let ((food (eats animal)))
        (push food (eaten animal))
        (format t "%-Feeding ~A with ~A" animal food)))
    (defmethod feed ((animal cat))
      (let ((food (eats animal)))
        (push food (eaten animal))
        (push 'milk (eaten animal))
        (format t "%-Feeding cat ~A with ~A and ~A" animal food 'milk)))
    (defvar *cat* (make-instance 'cat))
    (defvar *nellie* (make-instance 'elephant)))
*NELLIE*

CL-USER 46 > (feed *cat*)
Feeding cat #<CAT 6f35d4> with FISH and MILK
NIL

CL-USER 47 > (feed *nellie*)
Feeding #<ELEPHANT 71e7bc> with HAY
NIL

CL-USER 48 > (defadvice
    ((method feed (animal))
    after-feed :after)
    (animal)
    (format t "%-A has eaten -A" animal (eaten animal)))
NIL

CL-USER 49 > (defadvice
    ((method feed (cat))
    before-feed :before)
    (animal)
    (format t "%-Stroking -A" animal)
    (setf (how-happy animal) 'high))
NIL
6.5 Advising subfunctions

Subfunctions are functions that are defined inside the body of other functions rather than by top level defining forms like `defun`, `defmethod`, etc. To advise such a subfunction, call `defadvice` with a "subfunction dspec" of the form:

- `(subfunction sub-name parent-dspec)`
- `(flet sub-name parent-dspec)`
- `(labels sub-name parent-dspec)`

See “Subfunction dspecs” on page 85 for details. The rest of the `defadvice` form has the same effect as when advising ordinary functions.

The behavior when advising a subfunction is somewhat different from advising other functions.

- `defadvice` modifies the parent of the subfunction such that future execution of the code that creates the subfunction will create it advised.
- A subsequent call to `defadvice` with the same dspec and name re-modifies the parent, and also causes any of the subfunctions that were already created advised to change their "advice" behavior (as defined by the `defadvice` form) to the behavior specified by the latest call to `defadvice`.
- `remove-advice` with the same dspec and name returns the parent function to its original state, and switches off the "advice" for the subfunctions that were created "advised".
- Using `defadvice` again with the same dspec and name after `remove-advice` modifies the parent as in the first point above, but has no affect...
on any subfunction that was created by the parent before the call to
\texttt{remove-advice}.

Advising of subfunction works only for compiled code.

\subsection{Notes on subfunction names}

Anonymous lambdas are named by the compiler using integers.

You can find the dspec of a given subfunction by calling \texttt{object-dspec} on the
subfunction. You can also construct it from the printed representation of the
subfunction, which contains the sub-name and the parent-dspec.

A subfunction can be given an name using a \texttt{hcl:lambda-name} declaration
(see \texttt{declare}, page 651). If this is of the form \texttt{(subfunction sub-name)}, then
the dspec of the subfunction will be contain both \texttt{sub-name} and the correct par-
ent-dspec. However, if it has any other form, then dspec will be that name and
you will need to know the parent-dspec in order to construct the subfunction
dspec.

\section{Examples}

So far you have only seen examples of before and after pieces of advice. This
section contains some further examples. Suppose that you define a function
\texttt{alpha} that squares a number, and then decide that you intended to return the
reciprocal of the square instead. You might proceed as follows.

\begin{verbatim}
CL-USER 30 > (defun alpha (x) (* x x))
ALPHA
CL-USER 31 > (defadvice (alpha reciprocal :around) (num)
           (/ (call-next-advice num)))
NIL
CL-USER 32 > (alpha -5)
1/25
\end{verbatim}

First you change \texttt{alpha} to return the reciprocal of the square. Do this by defin-
ing an around method to take the reciprocal of the result produced by the next
piece of advice (which initially is the original definition). Now suppose that
you later decide that you would like `alpha` to return the sum of the squares of the reciprocals in a certain range. You can achieve this by adding an extra layer of around advice. This must iterate over the range required, summing the results obtained by the calls to the next piece of advice (which currently yields the reciprocal of the square of its argument).

```lisp
CL-USER 36 > (defadvice
(alpha sum-over-range :around)
(start end)
(loop for i from start upto end
  summing (call-next-advice i)))
NIL

CL-USER 37 > (alpha 2 5)
1669/3600
```

Note that `alpha` now behaves as a function requiring two arguments; the outer piece of around advice determines the external interface to the function, and uses the inner pieces of advice as it needs - in this case invoking the inner advice a variable number of times depending on the range specified. The use of the words “outer” and “inner” corresponds to earlier and later pieces of around advice in the ordering discussed above, but is more descriptive of their behavior.

You now realize that taking the reciprocal of zero gives an error. You decide that you wish to generate an error if `alpha` is called in such a way as to cause this, but that you want to generate the error yourself. You also decide to add a warning message for negative arguments. As you want these actions to be performed as the last (that is innermost) in the chain of around advice, you specify this in the call to `defadvice` by giving it a `:where` keyword with value `:end`.

```lisp
CL-USER 38 > (defadvice
(alpha sum-over-range :around)
(start end)
(loop for i from start upto end
  summing (call-next-advice i)))
NIL

CL-USER 39 > (alpha -2 5)
NIL
```

Note that `alpha` now ignores the negative argument and returns the sum of the squares of the reciprocals in the range specified.
Finally you decide to alter \texttt{alpha} yet again, this time to produce approximations to \( \pi \). \( \pi^2/6 \) is the sum of the reciprocals of the squares of all the positive integers. So you can generate an approximation to \( \pi \) using the sum of the reciprocals of the squares of the integers from one to some limit. (In fact this is not an efficient way of calculating \( \pi \), but it could be of interest.)
Next, try calling the following in turn:

(alpha 10.0)
(alpha 100.0)
(alpha 1000.0)
pi

Lastly, here is a simple example showing a use of advice with an &rest lambda list:

(defun foo (a b c)
 (print (list a b c)))

(defadvice (foo and-rest-advice :around) (&rest args)
 (format t "advice called with args ~S" args)
 (apply #'call-next-advice args))

6.7 Advice functions and macros

The main functions used for advice are introduced below. See the reference pages for full details.

The main macro used to define new pieces of advice is defadvice

Pieces of around advice should use call-next-advice to invoke the next piece of advice. As explained earlier this either calls the next piece of around advice (if one exists), or calls the combination of before advice, the original definition, and after advice. It may only be called from within the body of the around advice.

To remove a piece of advice, use the macro delete-advice or the function remove-advice.
The Advice Facility
The dspec system is the machinery underlying the way definitions are named in LispWorks. It supports program development by tracking the locations of definitions, and is also used in tracing and advising functions.

Dspects are not expected to work in runtimes delivered at a delivery level greater than 0.

This chapter explains the concepts underlying dspects and their use in tracking locations of definitions. For full details of the programming interface, see Chapter 36, “The DSPEC Package”.

### 7.1 Dspecs

Definition specifications, or *dspecs*, are a systematic way of naming definitions. The dspec system includes all kinds of definitions provided in LispWorks, and can be extended to include definers that you add.

Most named definitions are global, but local functions can have names, and some of the operations described here can be applied to them as well.

Here are three examples of dspects:

```
car
(setf car)
```
A dspec is simply a name: you can operate on it even if the thing named does not currently exist.

### 7.2 Forms of dspecs

A dspec is one of:

- A symbol
- A `setf` function name
- A list starting with a symbol naming the class of definition (method or `defstruct` for example).

A symbol which is used as a dspec always names a function or a macro.

\[(\text{setf \ foo})\] is a name for a setf function.

**Note:** nil is not a legal dspec, because it cannot have a function definition. Therefore when a dspec API returns nil, this should be interpreted in the usual way as “not found” or “not applicable”.

### 7.2.1 Canonical dspecs

Internally, dspecs are handled in the canonical form:

\[(\text{dspec-class primary-name . qualifiers})\]

where dspec-class in the canonical name of the class, and qualifiers is a proper list. primary-name is typically a symbol, but can be a list (in the case of a setf function) or a string (in the case of a package). The equality for canonical dspecs is `equal`.

As an example the general form of a `defmethod` dspec is:

\[(\text{defmethod name qualifiers (specializer*)})\]

name := symbol | (setf symbol)
qualifiers := qualifier | (qualifier qualifier*)
specializer := symbol | (eql object)
Functions in the dspec API accept non-canonical dspecs. All dspec functions, except \texttt{dspec:prettify-dspec}, \texttt{find-dspec-locations}, \texttt{name-definition-locations}, \texttt{dspec-definition-locations} and \texttt{find-name-locations} return canonical dspecs.

7.3 Dspec namespaces

Dspec classes are the namespaces for dspecs. Class names are often the same as the name of the defining form, though documentation types as defined for documentation are also used. See “Details of built-in dspec classes and aliases” on page 82 for a list of the classes.

7.3.1 Dspec classes

Dspec classes provide a set of handlers, to allow uniform handling of different types of definitions by other parts of the system, such as the editor and various browsers.

The most important handlers are \texttt{dspec-defined-p} and \texttt{dspec-undefiner} for testing if a dspec is currently defined and for undefining a dspec.

New dspec classes are defined using \texttt{define-dspec-class}.

Dspec classes can be subclassed. The top-level classes correspond to distinct global namespaces (such as \texttt{variable} for variables and constants and \texttt{function} for functions and macros), and at each level, all the subclasses are distinct from each other (but they do not have to form a complete partition of the superclass). See “Details of built-in dspec classes and aliases” on page 82 for the full hierarchy of system-provided classes.

You are allowed to define new top-level classes and subclass them, but you cannot add new subclasses to a system-provided class. However, see “Dspec aliases” on page 80 for how to add new ways of making existing definitions.

7.3.1.1 Complete example of a top-level dspec class

Define a \texttt{saved-value} object which has a name and a value:

```lisp
(defstruct saved-value
  name
  value)
```
The objects are defined using `def-saved-value` and stored on the plist of their name:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defmacro saved-value (name)
  `(saved-value-value (find-saved-value ',name)))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a function to retrieve the `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a dspec class for `def-saved-value` dspecs:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a `saved-value` object:

```lisp
(defun find-saved-value (name)
  (get name 'saved-value))
```
Note: this form parser for def-saved-value is not strictly necessary, because the system provides an implicit form parser which recognizes definitions beginning with "def".

7.3.1.2 Example of subclassing

This example is based on that in “Complete example of a top-level dspec class” on page 77.

Define a computed-saved-value object has a function to compute the value the first time:

```
(defstruct (computed-saved-value (:include saved-value))
  function)
```

saved-value objects are defined using def-computed-saved-value and stored on the plist of their name:

```
(defmacro def-computed-saved-value (name function)
  `(dspec:def (def-computed-saved-value ,name)
     (when (record-definition `(def-computed-saved-value ,',name)
                             (dspec:location)))
     (setf (get ',name 'saved-value)
           (make-computed-saved-value :name ',name :function ,function))
     ',name)))
```

Define a function to compute a computed-saved-value:

```
(defun ensure-saved-value-computed (name)
  (let ((saved-value (find-saved-value name)))
    (or (saved-value-value saved-value)
        (setf (saved-value-value saved-value)
              (funcall (computed-saved-value-function saved-value))))))
```

Define a macro to access a computed-saved-value:

```
(defmacro computed-saved-value (name)
  ~(ensure-saved-value-computed ,',name))
```

Define a dspec class for def-computed-saved-value dspecs:
For completeness, define a form parser that generates dspecs from forms:

(dspect:define-form-parser
  (def-computed-saved-value
  (:parser dspec:single-form-form-parser)))

Note: this form parser for def-computed-saved-value is not strictly necessary, because the implicit form parser will recognize definitions beginning with "def".

7.3.2 Dspec aliases

You can add new ways of making existing definitions and use the dspec system to track these definitions. This is what happens when your defining form expands into a system-provided form. The macro define-dspec-alias is used to inform the dspec system of this.

For example if your definer is:

(defmacro my-defun ((name &rest args) &body body)
  `(defun ,name ,args ,@body))

then you would define the form of dspecs for my-defun definitions like this:

(dspect:define-dspec-alias my-defun (name)
  `(defun ,name))

Note: in general you should not include the lambda list in the dspec, because it is not needed to locate the definition later.
Note: to make source location work you will also need a define-form-parser definition for my-defun. This is illustrated in “Using pre-defined form parsers” on page 91.

7.4 Types of relations between definitions

7.4.1 Functionally equivalent definers

When one definition form simply macroexpands into another, or otherwise has an identical effect as far as the dspec system is concerned, the dspec system should consider them variant forms of the same class.

Use define-dspec-alias to convert one definer to the other during canonicalization. A pre-defined example of this in LispWorks is defparameter and defvar. These cannot be distinguished (other than in the source code), so defparameter has been defined as a dspec alias for defvar. However, defvar and defconstant are distinct kinds of variable, since we can easily tell which type of definition is in effect by calling the function constantp. To define their dspecs, LispWorks creates a dspec class called variable and uses it as the superspace argument when defining the defvar and defconstant dspec classes.

As an explicit example, suppose you have a defining macro

```
(defmacro parameterdef  (value name)
  `(defparameter ,name ,value))
```

then

```
(dspec:define-dspec-alias parameterdef (value name)
  `(defparameter ,name))
```

would be a suitable appropriate alias definition. This define-dspec-alias form defines the dspec.

define-dspec-alias is like defmacro for dspecs, so it could be used to describe complicated conversions, as long as it can be done purely statically and totally in terms of existing dspecs. However, nothing more complicated than defparameter has been found necessary.
7.4.2 Grouping subdefinitions together

Some definition forms are macros that expand into a group of other definitions, for example **defstruct**. When the form is associated with a dspec class, the subdefinitions can be automatically recorded as being subforms of the new definition, by use of the **def** macro.

This means that the dspec system knows that the subdefinitions were inside the main definition (indeed, inside this particular form). Therefore

- Location queries can retrieve this information.
- The source location commands in the LispWorks IDE, when passed a subdefinition, know to search for the main definition given in the **def**.

**Note:** to make source location work you will also need a **define-form-parser** definition for the macro that expands into the **def**.

**Note:** **def** defines a relation between two particular definitions, for example (**defstruct foo**) and (**defun make-foo**), not between the two dspec classes.

7.4.3 Distributed definitions

Some definitions are additions to another class of definition, for example methods are additions to generic functions. We call these *distributed definitions*, consisting of "parts" and "the aggregate".

The primary name of a part gives the primary name of the aggregate it is a part of, and the qualifiers distinguish it from the other parts of the same aggregate. Only a part dspec may have qualifiers.

7.5 Details of built-in dspec classes and aliases

This section shows the dspec classes, subclasses and aliases provided by LispWorks. Subclasses are indented. Following the list of dspec classes are notes about some of these classes.

The system-defined dspec classes are:
7.5  Details of built-in dspec classes and aliases

COMPILER-MACRO (alias DEFINE-COMPILER-MACRO)
EDITOR:DEFCOMMAND (alias EDITOR:DEFINE-COMMAND-SYNONYM)
DEFINE-ACTION
DEFINE-ACTION-LIST
WIN32:DEFINE-DDE-CLIENT
WIN32:DEFINE-DDE-DISPATCH-TOPIC
DSPEC:DEFINE-DSPEC-ALIAS
EDITOR:DEFINE-EDITOR-VARIABLE (alias EDITOR:DEFINE-EDITOR-MODE-VARIABLE)
FLI:DEFINE-FOREIGN-CALLABLE
FLI:DEFINE-FOREIGN-TYPE (alias FLI:DEFINE-FOREIGN-CONVERTER)
DSPEC:DEFINE-FORM-_PARSER
CAPI:DEFINE-MENU
DEFSETF (aliases DEFINE-SETF-EXPANDER, DEFINE-SETF-METHOD)
DEFSYSTEM
FUNCTION
DEFGENERIC
DEFMACRO (alias DEFINE-MODIFY-MACRO)
DEFUN (alias SYSTEM:DEFUN-AND INLINE)
FLI:DEFINE-FOREIGN-VARIABLE
FLI:DEFINE-FOREIGN-FUNCTION (alias FLI:DEFINE-FOREIGN-FUNCALLABLE)
METHOD (alias DEFINE-METHOD)
METHOD-COMBINATION (alias DEFINE-METHOD-COMBINATION)
PACKAGE (alias DEFPACKAGE)
STRUCTURE (alias DEFSSTRUCT)
TYPE
DEFCLASS
CAPI:DEFINE-INTERFACE
CAPI:DEFINE-LAYOUT
DEFINE-CONDITION
STRUCTURE-CLASS
DEFTYPE
VARIABLE
DEF-SYMBOL-MACRO
DEF-CONSTANT
DEFVAR (aliases DEFINEGLOBAL-PARAMETER, DEFINEGLOBAL-VARIABLE, DEFINEPARAMETER)

Further dspec classes are defined by modules such as com (on Microsoft Windows), kw and sql.

The canonical form of a symbol dspec is (function symbol) and the canonical form of a setf function name dspec is (function (setf symbol)).
7.5.1 Function dspecs

A function-dspec is a dspec that names a specific function. You can use a function-dspec when you need to specify a function by name, for example in trace, defadvice, and set-up-profiler.

A function-dspec can be either a symbol, a list of the form \((\texttt{setf symbol})\), or any dspec with a class that is a "function" class, that is function or any of the classes listed above under function. It can also be a method dspec as described in “CLOS dspec classes” on page 84 or a subfunction dspec as described in “Subfunction dspecs” on page 85.

7.5.2 CLOS dspec classes

defgeneric and method can handle standard-generic-function and standard-method.

The canonical form of a defgeneric dspec is (defgeneric generic-function-name).

The canonical dspec of a method is:

\[(\texttt{method function-name [method-qualifier] (parameter-specializer-name*)})\]

Where function-name, method-qualifier and parameter-specializer-name match the ones in the defmethod form.

Each parameter-specializer-name must match the corresponding specializer for all the required parameters of the method. If a parameter is not specialized in the defmethod form then its parameter-specializer-name needs to be given as \(t\).

For example, a method that is defined by:

\[(\texttt{defmethod a-method ((arg1 cons) arg2 &optional arg3) \ldots})\]

has a dspec:

\[(\texttt{method a-method (cons t)})\]

and a method defined like this:

\[(\texttt{defmethod initialize-instance :after ((a my-class) &key key1 key2) \ldots})\]
7.6 Subfunction dspecs

has a dspec:

\[(\text{method } \text{initialize-instance} : \text{after} (\text{my-class}))\]

7.5.3 Part Classes

\textit{method} is a part class for \textit{defgeneric}.

\textit{compiler-macro} is a part class for \textit{function}.

7.5.4 Foreign callable dspecs

For \texttt{fli:define-foreign-callable} the canonical name is the foreign name, with any machine-specific prefixes omitted.

7.6 Subfunction dspecs

For some purposes, most usefully \textit{trace} and \textit{defadvice}, LispWorks allows dspecs that do not name a global definition, but a local function. These are of the form:

\[(\text{subfunction } \text{sub-name} \text{ parent-dspec})\]

where \textit{parent-dspec} is another dspec (possibly a subfunction dspec itself). For \texttt{flet} and \texttt{labels}, it is also possible to use the form:

\[(\text{flet } \text{sub-name} \text{ parent-dspec})\]

An alias for \texttt{(subfunction (flet sub-name) parent-dspec)}.

or:

\[(\text{labels } \text{sub-name} \text{ parent-dspec})\]

An alias for \texttt{(subfunction (labels sub-name) parent-dspec)}.

\textit{sub-name} is the name of the subfunction inside the parent, which by default is determined as follows:

- For subfunctions defined by \texttt{flet} and \texttt{labels}, the name is a two element list of the form \texttt{(flet function-name)} or \texttt{(labels function-name)},
where `function-name` is the function name in the `flet/labels` definition.

- For anonymous lambdas, the compiler names the subfunctions within each parent function by small, increasing integers starting from 1.

You can override the default name by using the LispWorks-specific `hcl:lambda-name` declaration (see `declare`). Note that you should use the form

```
(declare (lambda-name (subfunction sub-name))
```

to get a name that is useful for debugging. If you do not use `subfunction`, then the debugger cannot find the source for function.

Notes:

- The `sub-name` of the subfunction is not used by the dspec system to search its databases, so can be anything.

- Source level debugging does not use the `sub-name` of the subfunction, but does need to be able to find the definition of `parent-dspec`.

- `trace` and `defadvice` search for the function by comparing (using `equal`) the subfunction name in the dspec to the name of each subfunction in the parent function.

- A subfunction dspec can be canonicalized and prettified or passed as an argument to `dspec-definition-locations` (which will find where parent is defined).

- Additionally, pseudo-dspecs like this are allowed for top-level forms:

```
(top-level-form (location tlf))
```

`location` is an atomic location (not containing `:inside`) and `tlf` identifies the top-level form within that location. These are used as parent dspecs in subfunction dspecs and `:inside` locations. These dspecs can be canonicalized and prettified, and can be returned as dspecs from the location finders.
7.7 Tracking definitions

The dspec system is used to keep track of global definitions in many ways, and global definition macros usually tell the dspec system when the definition changes.

The main purpose of the system is to keep track of where the definition was located, but it also allows fine-tuned control of redefinitions.

7.7.1 Locations

Locations are mainly something the dspec system just stores and retrieves. :inside locations are used to describe definitions located as subforms of other definitions.

:inside locations are usually not explicitly specified, but arise as a result of having two nested definitions, both of which use the def and location macros to handle the name and location info.

The types of locations and their meanings are:

- A pathname: A definition existed in the file named or an editor buffer with that name.
- The keyword :listener: A definition was executed interactively in the listener or an editor buffer not associated with a file.
- The keyword :unknown: A definition was found in the image (these are entered when a location query does not find any information already in the database).
- The keyword :implicit: A definition for a part was recorded, but no information exists for the aggregate.

7.7.2 Recording definitions and redefinition checking

The location information is entered into the database when the definition is executed, by the defining function calling record-definition.
7 Dspecs: Tools for Handling Definitions

*record-definition* performs various checks, and returns true or false depending on whether the definition was allowed or not. In particular, it checks whether the same name has already been defined in a different location and if so a warning or error can be signaled. See *record-definition*, page 817 for details.

### 7.7.2.1 Use of record-definition

You should not usually call *record-definition*, since all the system-provided definers call it.

However, for new classes of definition which you add with define-dspec-class, you should call *record-definition* for dspecs in their new classes, as shown in “Complete example of a top-level dspec class” on page 77.

### 7.7.2.2 Protecting packages

LispWorks has a mechanism for protecting packages against defining any of their external symbols. By default, all the LispWorks implementation packages are protected. This is configurable by the variables *packages-for-warn-on-redefinition* and *handle-warn-on-redefinition*.

The protection is useful because it is relatively easy to redefine an external symbol by mistake, and it leads to undefined behavior which is difficult to debug. However, in some circumstances you may want to force such definition. In this case, you can rebind either of *packages-for-warn-on-redefinition* or *handle-warn-on-redefinition* around the definition to avoid the error.

You can also protect your packages by adding their names to the global value of *packages-for-warn-on-redefinition*.

### 7.7.3 Source level debugging and stepping

With suitable compilation options (see toggle-source-debugging), the LispWorks debugger will automatically identify the exact subform in the source code for each stack frame. In addition, the Stepper tool in the LispWorks IDE can step subforms in the source code.
This also works for a subform that occurs within a macro expansion, provided that the subform is eq to the original subform in the call to the macro. In the rare case where a macro copied a subform, making it non-eq, you can use the replacement-source-form macro to indicate which original subform should be identified as the source code for the new form.

7.8 Finding locations

There are two ways of retrieving location information for definitions in the running LispWorks image:

- query for a dspec using dspec-definition-locations, or
- query for a name in a given set of namespaces using name-definition-locations

The difference is that name queries will find the locations of all the part definitions as well as the definition named, whereas dspec queries will only find the locations for the definition named (there might be many if it has been redefined).

To provide for sub-definitions hidden in another definition, such as defstruct accessors, all location queries produce a list of pairs of dspecs and locations, each pair naming a definition within the corresponding location that contains the definition looked for. So a query for an accessor called foo-bar might produce the pair:

```
((defstruct foo) #P"/usr/users/hacker/hacks/hack.lisp")
```

7.9 Users of location information

To find location information for definitions made in the running image or recorded in a tags database or a tags file:

- query for a dspec using find-dspec-locations, or
- query for a name in a given set of namespaces using find-name-locations

The extent of the search is controlled by the value of the variable *active-finders*. 
For example, to obtain the locations of the definitions of foo across all dspec namespaces, call

```
(dspect:find-name-locations dspec:*dspec-classes* 'foo)
```

Another example of the use of find-name-locations is the LispWorks Editor tool's Find Definitions tab.

### 7.9.1 Finding definitions in the LispWorks editor

Returning to our example `parameterdef` definer

```
(defun parameterdef (value name)
  `(defparameter ,name ,value))
```

1. Load a file `foo.lisp` containing
   `(parameterdef 42 *foo*)`

2. Now use **Expression > Find Source** on the symbol `*foo*`. Notice that LispWorks knows which file the definition is in, but cannot find the defining top level form.

3. Also notice that the Definitions tab of the Editor tool does not display the definition of `*foo*`. This is because the Editor does not recognize `parameterdef` as a definer. When the LispWorks editor looks at the definitions in a buffer, it needs to know the dspecs that each defining form will generate when evaluated. You can tell the editor how to parse a defining form to generate the dspec by using `define-form-parser`.

4. Now evaluate these forms to associate a parser with `parameterdef` and inform the dspec system that `parameterdef` is another way of naming a `defparameter` dspec:

   ```lisp
   (dspec:define-form-parser parameterdef (value name)
   "(parameterdef ,name))

   (dspec:define-dspec-alias parameterdef (name)
   "(defparameter ,name))
   ``

5. Now use **Expression > Find Source** on the symbol `*foo*` again. Notice that the source of the definition of `*foo*` is displayed correctly in the text tab of the Editor tool, and that the Definitions tab displays the definition as
7.9 Users of location information

7.9.2 Using pre-defined form parsers

LispWorks provides form parsers `name-only-form-parser`, `single-form-parser` and `single-form-with-options-form-parser`. You can use `single-form-with-options-form-parser` as the parser for `my-defun` definitions (see “Dspec aliases” on page 80), like this:

```
(dspec:define-form-parser (my-defun
                         (:parser dspec:single-form-with-options-form-parser)))
```

This allows the Editor to locate definitions like:

```
(my-defun (foo x y)
  (+ x y))
```

You can identify the form parser defined for a dspec class using `get-form-parser`.

7.9.3 The editor’s implicit form parser

When testing your form parsers bear in mind that the LispWorks editor has an implicit form parser, independent of explicit parsers defined in the dspec system. It tries to parse a dspec from a top level form which is of length 2 or more and whose car has symbol name beginning with *DEF*. That is:

```
(defxyz name forms)
```

gets parsed as

```
(defxyz name)
```

which may be a dspec (and thus provides a match for the source location commands). This mechanism operates only when there’s no explicit parser defined for `defxyz`.

The editor’s implicit form parser is useful because it matches a common simple case. However it does not work for the `parameterdef` example, because that definer’s symbol name does not begin with *DEF*. 

```lisp
(parameterdef *foo*)
```
7.9.4 Reusing form parsers

The form parser established above was specifically for `parameterdef` forms. However if you have other definers of similar syntax - in this example, definers for which the name is the second subform - then you can define a form parser which can be associated with each of them, as follows:

```
(dspec:define-form-parser (name-second (:anonymous t))
  (value name)
  ~(,name-second ,name))
```

Note that the `name-second` variable is evaluated in the body of the parser. Supposing you have another defining macro `constantdef`:

```
(defmacro constantdef (value name)
  ~(defconstant ,name ,value))
```

then you can associate the same parser with both this and `parameterdef`:

```
(dspec:define-form-parser (parameterdef (:parser name-second-form-parser)))
(dspec:define-form-parser (constantdef (:parser name-second-form-parser)))
```

7.9.5 Example: defcondition

Suppose you have a macro based on `define-condition`:

```
(defmacro defcondition (&rest args)
  ~(define-condition ,@args))
```

When the following form is evaluated, the system records the dspec (define-condition foo):

```
(defcondition foo () ())
```

Two setups are needed to allow the editor to locate such a defining form. Firstly, this tells the system how to parse (defcondition ...) toplevel forms:

```
(dspec:define-form-parser
  (defcondition
   (:alias define-condition)))
```

So now:
7.9 Users of location information

(defun foo () ())

Secondly, this tells the system that (defcondition foo) is an alias for (define-condition foo).

With this, the editor would report "Cannot find (DEFINE-CONDITION FOO) in ...".

(defun foo () ())

just as if it were:

(defun foo () ())

7.9.6 Example: my-defmethod

Suppose you have a method definer my-defmethod:

(defun my-defmethod ((name &key doc)
  lambda-list
  &body body)
  `(defmethod ,name ,lambda-list ,@body))

Unlike function dspecs, method dspecs need to include the specialized argument types as well as the function name, so the alias and the parser both need to be more complex.

This causes the dspec to include the argument types:

(defun my-defmethod ((name &key doc)
  lambda-list)
  `(defmethod ,name ,lambda-list ,@body))

The dspec for method lambda lists is complicated, but you can invoke the defmethod parser in your my-defmethod parser, like this:

(defun my-defmethod ((name &key doc)
  lambda-list)
  `(defmethod ,name ,lambda-list ,@body))

```lisp
(dspec:define-dspec-alias my-defmethod (name &rest options)
  `(defmethod ,name ,&rest options))
```

The dspec parser for method lambda lists is complicated, but you can invoke the defmethod parser in your my-defmethod parser, like this:

```lisp
(dspec:define-dspec-alias my-defmethod (name &rest options)
  `(defmethod ,name ,&rest options))
```

The dspec parser for method lambda lists is complicated, but you can invoke the defmethod parser in your my-defmethod parser, like this:

```lisp
(dspec:define-dspec-alias my-defmethod (name &rest options)
  `(defmethod ,name ,&rest options))
```

The dspec parser for method lambda lists is complicated, but you can invoke the defmethod parser in your my-defmethod parser, like this:

```lisp
(dspec:define-dspec-alias my-defmethod (name &rest options)
  `(defmethod ,name ,&rest options))
```

The dspec parser for method lambda lists is complicated, but you can invoke the defmethod parser in your my-defmethod parser, like this:

```lisp
(dspec:define-dspec-alias my-defmethod (name &rest options)
  `(defmethod ,name ,&rest options))
```

The dspec parser for method lambda lists is complicated, but you can invoke the defmethod parser in your my-defmethod parser, like this:

```lisp
(dspec:define-dspec-alias my-defmethod (name &rest options)
  `(defmethod ,name ,&rest options))
```
Now this definition can be located:

```lisp
(my-defmethod (bar :doc "bar documentation") (x y)
  (foo x y))
```

just as if it were:

```lisp
(defmethod bar (x y)
  (foo x y))
```
Action Lists

Action-lists are a unified approach to various different mechanisms for running initializations, or “hook” functions at various points during the life of the system. They provide central gathering points for applications to trigger on system-wide events such as start-up, disk-save, and so on.

An action-list is a tagged list of data, to be executed (in some sense) in sequence whenever the circumstance identified by its tag occurs. It is expected that whatever code detects or causes the circumstance will take care of running the action-list.

An execution-function can be specified for the action-list when it is created. Otherwise, the default behavior is to treat the data of each action as a callable and apply it to any additional arguments specified at execution time. At its simplest, an action-list emulates `(map nil 'funcall).

Names of action-lists and action-items are general lisp objects, compared with `equalp. This allows strings and other objects to be used as unique identifiers.

Actions can be specified to depend on other actions; when defining an action-item, you can say that it must be before or after other action-items using the `:before and `:after keywords. Aside from that, actions are assumed to have no dependencies, and no order of execution should be counted on for the actions in a list.
8 Action Lists

You can (and are encouraged to) specify a documentation string for action-lists or action-items.

In addition you can create action-lists that are not registered globally. This allows applications to have disembodied action lists for their own internal purposes. The other action-list functions allow an action-list to be passed in instead of a name, to accommodate this.

8.1 Defining action lists and actions

Action lists are defined using the define-action-list macro, and are undefined using the undefine-action-list. It is also possible to make unnamed, unregistered lists using make-unregistered-action-list.

When defining an action-list, the user may provide an associated execution-function. When executing the action-list, this user-defined execution-function is used instead of the default execution-function, to map over and “execute” the action-list’s action-items. The macro with-action-list-mapping provides facilities to map over action-items (that is, their corresponding “data”). In addition, the macro with-action-list-mapping provides a simple mechanism to trap errors and print warnings while executing each action-item.

Actions are added to an action list using define-action, and are removed using undefine-action.

8.2 Exception handling variables

Three global variables control the handling of exceptions in action list and action item operations.

The variable *handle-existing-action-list* controls the behavior of define-action-list when the action list already exists. It allows you to control independently both:

- whether you are notified, and
- whether the action list gets redefined

The variable *handle-existing-action-in-action-list* controls the behavior of define-action when the action already exists in the given action-list. It allows you to control independently both:
whether you are notified, and
whether the action item gets redefined

The variable `*handle-missing-action-list*` specifies behavior when one of `undefined-action-list, print-actions, execute-actions, define-action` and `undefined-action` is called on a missing action-list. By default, an error is signaled, but you can make it warn or ignore instead.

The variable `*handle-missing-action-in-action-list*` specifies behavior when you attempt to undefine a missing action. By default, a warning is signaled, but you can make it signal error, or ignore, instead.

## 8.3 Other variables

The variable `*default-action-list-sort-time*` specifies when actions in action-lists are sorted. By default actions are sorted at the time of execution of the action list, but you can cause them to be sorted at action definition time instead.

See `define-action-list` for an explanation of ordering specifiers.

## 8.4 Diagnostic utilities

Two diagnostic functions are provided:

- `print-actions` prints out the actions on a specified action list
- `print-action-lists` prints a list of all the defined action lists

## 8.5 Examples

This example illustrates “typical” use of action lists. The `define-action` forms might be scattered across several files (`mail-utilities.lisp`, `caffeine.lisp`, and so on). Each of the functions, such as `read-mail`, `dont-panic`, and so on, take one argument: `hassled-p`.

```
(in-package "CL-USER")
```
(define-action-list "On arrival at office"
  :documentation "Things to do in the morning"
  :dummy-actions '("Look busy")
  :default-order '(:before "Look busy"))

(define-action "On arrival at office" "Read mail" 'read-mail)

(define-action "On arrival at office" "Greet co-workers"
  'say-hello)

(define-action "On arrival at office" "Drink much coffee"
  'wake-up:after "Locate coffee machine")

(define-action "On arrival at office" "Locate coffee machine"
  'dont-panic)

(defun my-morning (hassled-p Monday-p)
  (execute-actions ("On arrival at office"
     :ignore-errors-p Monday-p
     hassled-p)
  <rest of my-morning code goes here>)

This example illustrates use of execution-functions and post-processing

(in-package "CL-USER")

Here are the implementation details, which are hidden from the “user”.

(defstruct (thing (:constructor make-thing (name number)))
  name
  number)

(defvar *things*
  (make-unregistered-action-list :sort-time :define-action
    :execution-function 'act-on-things))

(defun do-things (function &optional post-process)
  (execute-actions (*things* :post-process post-process
    function)))

(defun act-on-things (things other-args-list &key post-process)
  (with-action-list-mapping
    (things ignore thing post-process)
    (destructuring-bind
      (function) other-args-list
      (funcall function thing)))))

The interface is given below. The internals of the mapping mechanism are hid-

98
(defmacro define-thing (name number)
  (with-unique-names (thing)
    ~(let ((,thing (make-thing ,name ,number)))
      (define-action *things* ',name ,thing))))

(defmacro undefine-thing (name)
  ~(undefine-action *things* ,name))

(defun find-thing (name)
  (do-things #'(lambda (thing)
        (and (equal name (thing-name thing))
            thing))
           :or))

(defun add-things ()
  (reduce '+ (do-things 'thing-number :collect)))

8.6 Standard Action Lists

The following action lists are defined in LispWorks as shipped:

"When starting image" - Actions to be executed upon image startup.

"Confirm when quitting image" - Actions to be executed before the image quits. Every action must return non-nil as its first value, otherwise the quit will be aborted once the actions are complete.

"When quitting image" - Actions to be executed when the image quits, after success of the "Confirm when quitting image" actions.

"Initialize LispWorks Tools" - Things to do when the LispWorks IDE starts on a screen. You may customize your environment startup by defining actions on it.

"Delivery Actions" - Actions to be executed when doing delivery. Actions on this list are executed in a 'normal' environment. See the Delivery User Guide for an example action item.

"Save Session Before" - Actions executed before saving a session. See save-current-session for details.

"Save Session After" - Actions executed after saving a session and redisplaying all the windows. These actions are executed both in the saving image and in the saved image when restarted. See save-current-session for details.
8 Action Lists
The Compiler

The compiler translates Lisp forms and source files into binary code for the host machine. A compiled Lisp function, for instance, is a sequence of machine instructions that directly execute the actions the evaluator would perform in interpreting an application of the original source lambda expression. Where possible the behaviors of compiled and interpreted versions of the same Lisp function are identical. Unfortunately the definition of the Common Lisp language results in certain unavoidable exceptions to this rule. The compiler, for instance, must macroexpand the source before translating it; any side effects of macro-expansion happen only once, at compile time.

By using declarations, you can advise the compiler of the types of variables local to a function or shared across an application. For example, numeric operations on a variable declared as a single-float can be compiled as direct floating-point operations, without the need to check the type at execution time. You can also control the relative emphasis the compiler places on efficiency (speed and space), safety (type checking) and support for debugging. By default the compiler produces code that performs all the necessary type checking and includes code to recover from errors. It is especially important that the type declarations be correct when compiling with a safety level less than 3 (see later in this chapter for more details).

When compiling a Lisp source file, the compiler produces its output in a format that is much faster to load than textual Lisp source — the “fasl” or “fast-
load” form. Fasl files contain arbitrary Common Lisp objects in a pre-digested form. They are loaded without needing to use the expensive read function. A series of “fasl-loader” routines built into LispWorks interpret the contents of fasl files, building the appropriate objects and structures in such a way that objects that were eq before fasl-dumping are created eq when fasl-loaded.

Fasl files are given pathname extensions that reflect the target processor they were compiled for; as the fasl files contain processor specific instruction sequences it is essential that the loader be able to distinguish between files compiled for different targets. These pathname extensions always end in “fasl”. See compile-file for details of all the possible fasl file extensions.

### 9.1 Compiling a function

The function compile takes a symbol as its first argument, and an interpreted function definition (a lambda expression) as its second, optional, argument. It compiles the definition and installs the resultant code as the symbol-function of the symbol (unless the symbol was nil). If the definition is omitted then the current symbol-function of the symbol is used. Below are some examples:

```lisp
CL-USER 3 > (compile (defun fred (a b)
    (dotimes (n a) (funcall b)))

; FRED
FRED
NIL
NIL

CL-USER 4 > (funcall (compile nil '(lambda (n)
    (* n n)) 7)

; NIL
49

CL-USER 5 > (compile 'ident-fun '(lambda (x) x))

; IDENT-FUN
IDENT-FUN
NIL
NIL
```
9.2 Compiling a source file

The function `compile-file` takes a pathname as its argument and compiles all the forms in the file, producing a corresponding fasl file (with pathname derived from the source pathname). Any side effects in the source file are only felt once the compiled file is subsequently loaded. Many proclamations, for example, are not visible at compile time. The special form `eval-when` can be used to force such side effects to take effect at the time of compilation, rather than loading.

9.3 Compiling a form

To compile an arbitrary form `form` (as opposed to a function), call

```
(compile form)
```

This compiles `form` as if by `compile-file` but without any file related processing and does it in-memory, so it has also the same effect as loading. This has a similar effect to compiling a definition in the LispWorks Editor tool, except that there is no source recording.

Using `compile` this way is especially useful if you need to dynamically define something that is normally defined by a top level or, for example `kw:defrule`.

9.4 How the compiler works

Conceptually the compiler can be viewed as performing a series of separate passes.

- In the first pass the source code is macro expanded in the appropriate macro environment.
- A series of source to source optimizing transformations are performed to simplify the source tree. Type declarations are used to select specialized, efficient versions of low level functions.
- A graph is generated from the source tree. The structure of the graph reflects the flow of control in the tree. The nodes of the graph contain blocks of intermediate code for an abstract machine with byte addressing and an infinite set of registers. Register allocation is performed
based on data flow analysis and machine specific rules concerning live ranges across code fragments.

- The blocks of intermediate code are translated into a single linear sequence of target machine code through a process of template matching.
- Finally the relative branch instructions are “fixed up” to point to the correct locations in the code sequence.

The compiler is in fact much more complex than this model might suggest. Machine specific optimizations, for example, can be included in any of the passes. The distinction between passes is also not as simple as that listed above. However, this description is sufficient to allow the programmer to make optimal use of the compiler.

### 9.5 Compiler control

There are ways to control the nature of compiled code via the `declare` special form and `proclaim` function. See “Declare, proclaim, and declaim” on page 109 for fuller discussion of these two forms.

In particular there are a set of optimize qualities which take integral values from 0 to 3. These control the trade-offs between size, speed, retention of debug information, optimizations and safety (that is, type checks) in the resulting code, and also compilation time. For example:

```
(proclaim '(optimize (speed 3) (safety 0) (debug 0)))
```

tells the compiler to concentrate on code speed rather than anything else, and

```
(proclaim '(optimize (safety 3)))
```

ensures that the compiler never takes liberties with Lisp semantics and produces code that checks for every kind of error that can be signaled.

The important declarations to the compiler are type declarations and optimize declarations. To declare that the type of the value of a variable can be relied upon to be unchanging (and hence allow the compiler to omit various checks in the code), say:

```
(declare (type the-type variable * ))
```
Optimize declarations have various qualities, and these take values from 0 to 3. The names are `safety`, `fixnum-safety`, `float`, `sys:interruptable`, `debug`, `speed`, `compilation-speed`, and `space`.

Most of the qualities default to 1 (but `safety` and `fixnum-safety` default to 3 and `interruptable` defaults to 0). You can either associate an optimize quality with a new value (with local lexical scope if in `declare`, and global scope if `proclaim`), or just give it by itself, which implies the value 3 (taken to mean “maximum” in some loose sense).

Thus you ensure code is at maximum safety by:

```lisp
(proclaim '(optimize (safety 3)))
```

or

```lisp
(proclaim '(optimize safety))
```

and reduce debugging information to a minimum by

```lisp
(proclaim '(optimize (debug 0)))
```

Normally code is interruptible, but when aiming for maximum speed and minimum safety and debug information code is not interruptible unless you ensure it thus:

```lisp
(proclaim '(optimize (debug 0) (safety 0) (speed 3) interruptable))
```

The levels of `safety` have the following implications:

- 0 implies no type checking upon reading or writing from defstructs, arrays and objects in general, nor any checking of array index bounds.
- 1 implies no type checking upon reading from defstructs, arrays and objects in general, nor any checking of array index bounds when reading. However, array index bounds are checked when writing.
- 2 implies type checking when writing, but not when reading. Other than this the compiler generates generally safe code, but allows `type` and `fixnum-safety` declarations to take effect. Array index bounds are checked for both reading and writing.
- 3 (default) implies complete type and bounds checking, and disallows `fixnum-safety` and `type` declarations from taking any effect.
The levels of **fixnum-safety** have the following implications:

- 0 implies no type checking of arguments to numeric operations, which are assumed to be fixnums. Also the result is assumed, without checking, to not overflow - this level means single machine instructions can be generated for most common integer operations, but risks generating values that may confuse the garbage collector.

- 1 implies that numeric operations do not check their argument types (assumed fixnum), but do signal an error if the result would have been out of range.

- 2 implies that numeric operations signal an error if their arguments are non-fixnum, and also check for overflow.

- 3 (default) implies complete conformance to the semantics of Common Lisp numbers, so that types other than integers are handled in compiled code.

Additionally if the level of **float** (really this should be called “float-safety”) is 0 then the compiler reduces allocation during float calculations.

The effects of combining these qualities is summarized below:

**Table 9.1** Combining debug and safety levels in the compiler

<table>
<thead>
<tr>
<th>Keyword settings</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>safety=0</td>
<td>Array access optimizations</td>
</tr>
<tr>
<td>debug&gt;0</td>
<td>Dumps symbol names for arglist</td>
</tr>
<tr>
<td>debug&gt;=2</td>
<td>Ensure debugger knows values of args (and variables when source level debugging is on) and can find the exact subform in the Editor.</td>
</tr>
<tr>
<td>debug&lt;1</td>
<td>Does not generate any debug info at all</td>
</tr>
<tr>
<td>debug=3</td>
<td>Avoids <strong>make-instance</strong> and <strong>find-class</strong> optimizations</td>
</tr>
<tr>
<td>debug=3</td>
<td>Avoids <strong>gethash</strong> and <strong>puthash</strong> optimizations</td>
</tr>
<tr>
<td>debug=3</td>
<td>Avoids <strong>ldb</strong> and <strong>dpb</strong> optimizations</td>
</tr>
</tbody>
</table>
9.5 Compiler control

Table 9.1 Combining debug and safety levels in the compiler

<table>
<thead>
<tr>
<th>Keyword settings</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>debug=3</strong></td>
<td>Avoids an optimization to last</td>
</tr>
<tr>
<td><strong>safety&gt;1</strong></td>
<td>Be careful when multiple value counts are wrong</td>
</tr>
<tr>
<td><strong>safety&lt;1</strong></td>
<td>Do not check array indices during write</td>
</tr>
<tr>
<td><strong>safety&lt;2</strong></td>
<td>Do not check array indices during read</td>
</tr>
<tr>
<td><strong>speed&gt;space</strong></td>
<td>Inline map functions (unless debug&gt;2)</td>
</tr>
<tr>
<td><strong>debug&lt;=2</strong></td>
<td>Optimize (merge) tail calls</td>
</tr>
<tr>
<td><strong>debug&lt;2 and safety&lt;2</strong></td>
<td>Self calls</td>
</tr>
<tr>
<td><strong>safety&gt;=2</strong></td>
<td>Check get special</td>
</tr>
<tr>
<td><strong>safety&lt;2</strong></td>
<td>Do not check types during write</td>
</tr>
<tr>
<td><strong>safety&lt;3</strong></td>
<td>Do not check types during read</td>
</tr>
<tr>
<td><strong>safety&gt;=1</strong></td>
<td>Check structure access</td>
</tr>
<tr>
<td><strong>safety&lt;1</strong></td>
<td>Inline structure readers, with no type check</td>
</tr>
<tr>
<td><strong>safety=0</strong></td>
<td>Inline structure writers, with no type check</td>
</tr>
<tr>
<td><strong>safety&lt;1 or interruptible&gt;0</strong></td>
<td>Check stack overflow</td>
</tr>
<tr>
<td><strong>safety&gt;1</strong></td>
<td>Ensures the thing being funcalled is a function</td>
</tr>
<tr>
<td><strong>safety&lt;3 and fixnum-safety=2</strong></td>
<td>Fixnum-only arithmetic with errors for non fixnum arguments.</td>
</tr>
<tr>
<td><strong>safety&lt;3 and fixnum-safety=1</strong></td>
<td>No fixnum overflow checks</td>
</tr>
<tr>
<td><strong>safety&lt;3 and fixnum-safety=0</strong></td>
<td>No fixnum arithmetic checks at all</td>
</tr>
<tr>
<td><strong>safety&gt;2</strong></td>
<td>char= checks for arguments of type character</td>
</tr>
<tr>
<td><strong>safety&gt;=2</strong></td>
<td>Ensures symbols in progv</td>
</tr>
</tbody>
</table>
The other optimize qualities are: speed — the attention to fast code, space — the degree of compactness, compilation-speed — speed of compilation, interruptable — whether code must be interruptible when unsafe.

Note that if you compile code with a low level of safety, you may get segmentation violations if the code is incorrect (for example, if type checking is turned off and you supply incorrect types). You can check this by interpreting the code rather than compiling it.

### 9.5.1 Examples of compiler control

The following function, compiled with safety = 2, does not check the type of its argument because it merely reads:

```lisp
(defun foo (x)
  (declare (optimize (safety 2)))
  (car x))
```

However the following function, also compiled with safety = 2, does check the type of its argument because it writes:

```lisp
(defun set-foo (x y)
  (declare (optimize (safety 2)))
  (setf (car x) y))
```

As another example, interpreted code and code compiled at at low safety does not check type declarations. To make LispWorks check declarations, you need to compile your code after doing:

```lisp
(declaim (optimize (safety 3) (debug 3)))
```
This last example shows how to copy efficiently bytes from a typed-aref vector (see make-typed-aref-vector) to an (unsigned-byte 8) array. type and safety declarations cause the compiler to inline the code that deals specifically with (unsigned-byte 8). This code was developed after an application was found to have a bottleneck in the original version of this function:

(defun copy-typed-aref-vector-to-byte-vector
  (byte-vector typed-vector length)
  (declare (optimize (safety 0)))
  (type (simple-array (unsigned-byte 8) 1) byte-vector)
  (fixnum length))
(dotimes (index length)
  (declare (type fixnum index))
  (setf (aref byte-vector index)
    (sys:typed-aref '(unsigned-byte 8)
      typed-vector index))))

9.6 Declare, proclaim, and declaim

The special form declare is used to:

- declare Lisp variables as “special”, which affects the semantics of the appropriate bindings of the variables, or
- help the system (in reality the compiler) run your Lisp code faster, or
- make the code run with more sophisticated debugging options, or
- help you optimize your code

declare behaves computationally as if it is not present (other than to affect the semantics), and is only allowed in certain contexts, such as after the variable list in a let, do, defun and so on. Consult the syntax definition of each special form to see if it accepts declare forms.

For the details, including some LispWorks extensions to Common Lisp, see the reference entry for declare.

The function proclaim parses declarations in a specified list and then puts their semantics and advice into global effect. This can be useful when compiling a file for speedy execution, since a proclamation such as:

(proclaim '(optimize (speed 3) (space 0) (debug 0)))
causes the rest of the file to be compiled with these optimization levels in effect. (A lengthier way to do this is to make appropriate declarations in every function in the file.) Below are some more examples:

(proclaim '(special *fred*))
(proclaim '(type single-float x y z))
(proclaim '(optimize (safety 0) (speed 3)))

Do not forget to quote the argument list if it is a constant list. This form:
(proclaim (special x))

attempts to call function special.

declare is a macro equivalent to proclaim.

9.6.1 Naming conventions

Exercise caution if you declare or proclaim variables to be special without regard to the naming convention that surrounds their names with asterisks.

9.7 Optimizing your code

Careful use of the compiler optimize qualities described above or special declarations may significantly improve the performance of your code. However it is not recommended that you simply experiment with the effect of adding declarations. It is more productive to work systematically:

1. Use the Profiler, described in Chapter 12, “The Profiler”, to analyze your application’s performance and identify bottlenecks, then

2. Consider whether re-writing of parts of your source code would improve efficiency at the bottlenecks, and

3. Use :explain declarations to make the compiler generate optimization hints, and

4. (In SMP LispWorks) use analysing-special-variables-usage to report on symbols proclaimed special, and

5. Consider adding suitable declarations as described in this chapter to improve efficiency at the bottlenecks.
The most important tool for speeding up programs is the Profiler. You use the profiler to find the bottlenecks in the program, and then optimize these bottlenecks by helping the compiler to produce better code.

The remainder of this section describes some specific ways to produce efficient compiled code with LispWorks.

### 9.7.1 Compiler optimization hints

You can make the compiler print messages which will help you to optimize your code. You add suitable `:explain` declarations, recompile the code, and check the output.

The full syntax of the `:explain` declaration is documented in the reference entry for `declare`.

Various keywords allows you to see information about compiler transformations depending on type information, allocation of floats and bignums, floating point variables, function calls, argument types and so on. Here is a simple example:

```lisp
(defun foo (arg)
  (declare (:explain :variables) (optimize (float 0)))
  (let* ((double-arg (coerce arg 'double-float))
         (next (+ double-arg 1d0))
         (other (* double-arg 1/2)))
    (values next other)))

;;- Variables with non-floating point types: 
;;- ARG OTHER 
;;- Variables with floating point types: 
;;- DOUBLE-ARG NEXT
```

**Note:** the LispWorks IDE allows you to distinguish compiler optimization hints from the other output of compilation, and also helps you to locate quickly the source of each hint. For more information see the chapter “The Output Browser” in the *LispWorks IDE User Guide*.

### 9.7.2 Fast integer arithmetic

You can arrange for compiled code to perform optimal raw 32-bit arithmetic, and additionally in 64-bit LispWorks, optimal raw 64-bit arithmetic.
For all the details, see “Fast 32-bit arithmetic” on page 472 and “Fast 64-bit arithmetic” on page 474.

9.7.3 Floating point optimization

The declaration float allows generation of more efficient code using float numbers. It reduces allocation during float calculations. It is best used with safety 0. That is, you declare (optimize (float 0) (safety 0)) as in this example:

(progn
  (setf a
    (make-array 1000
      :initial-element 1D0
      :element-type 'double-float))
  nil ; to avoid printing the large array)

(compile
  (defun test (a)
    (declare (optimize (speed 3) (safety 0) (float 0)))
    (declare (type (simple-array double-float (1000)) a))
    (let ((sum 0D0))
      (declare (type double-float sum))
      (dotimes (i 1000)
        (incf sum (the double-float (aref a i))))
      sum))

time (test a))
=>
Timing the evaluation of (TEST A)
user time    = 0.000
system time  = 0.000
Elapsed time = 0:00:00
Allocation   = 16 bytes standard / 0 bytes conses
0 Page faults

Note: In some cases, the operations cannot be fully optimized with float 0, which can cause the compiled code to be larger because the unboxing and boxing of floats will be inline.
9.7 Optimizing your code

9.7.4 Tail call optimization

In 64-bit LispWorks and on x86 platforms the compiler optimizes tail calls unless

1. The compiler optimize quality debug is 3, or

2. There is something with dynamic scope on the stack, such as a special binding, a catch or cl:dynamic-extent allocation (so it is not really a tail call)

On all other platforms the compiler optimizes tail calls unless 1.) or 2.) above apply, or

3. The call has more than 4 arguments and this is more than the number of fixed (not &optional/&rest/&key) parameters in the calling function.

4. The call has more than 4 arguments and the calling function has &rest/&key parameters.

9.7.5 Usage of special variables

The declaration cl:special specifies that a variable is special, that is it does not have lexical scope. This covers two cases: if the variable is bound in the dynamic environment (for example by let or let*), then the value of that binding is used; otherwise the value in the global environment is used, if any. An error is signaled in safe code if there is no value in either environment.

Whensetqis used with a variable, the value in the dynamic environment is modified if the variable is bound in the dynamic environment, otherwise the value in the global environment is modified. Dynamic variables can have a different value in each thread because each thread has its own dynamic environment. The global environment is shared between all threads.

In SMP LispWorks access to special variables (excluding constants) is a little slower than in non-SMP LispWorks. It can be made to run faster by declarations of the symbol, normally by using by proclaim or declare.

The speedup will be pretty small overall in most cases, because access to specials is usually a small part of a program. However, if the Profiler identifies some piece of code as a bottleneck, you will want to optimize it, and your optimizations may include proclamation of some variable as global or dynamic.
The three declarations described in this section are extensions to Common Lisp. All declare the symbol to be `cl:special`, along with other information. These three declarations are mutually exclusive between themselves and `cl:special`. That is, declaring a symbol with any of these declarations eliminates the other declaration:

- **hcl:special-global** declares that the symbol is never bound in the dynamic environment.

  In SMP LispWorks the compiler signals error if it detects that a symbol declared as `hcl:special-global` will be bound in the dynamic environment, and at run time it also signals an error.

  In non-SMP LispWorks the compiler gives an error, but there is no run time check. The run time behavior is the same as `cl:special`, with all accesses to the symbol in low safety.

  `hcl:special-global` is very useful, and because of the checks it is reasonably safe. It is useful not only for speed, but also to guard against unintentionally binding variables that should not be bound.

  See also `defglobal-parameter`.

- **hcl:special-dynamic** declares that the symbol is always bound in the dynamic environment when it is accessed.

  In high safety code accessing the symbol when it is not bound in the dynamic environment signals an error. In low safety code it may result in unpredictable behavior.

  In non-SMP LispWorks the only effect of this declaration is to make all access to the variable low safety.

  `hcl:special-dynamic` is useful, but because it can lead to unpredictable behavior you need to ensure that you test your program in high safety when you use it.

- **hcl:special-fast-access** declares that a symbol should be "fast access".
The semantics of the declaration is the same as \texttt{cl:special}, except that access to the variable is low safety. In addition, the compiler compiles access to the symbol in a way that speeds up the access, but also introduces a tiny reduction in the speed of the whole system. The balance between these effects is not obvious.

It is not obvious where \texttt{hcl:special-fast-access} is useful. If you can ensure that the symbol is always bound or never bound then \texttt{hcl:special-dynamic} or \texttt{hcl:special-global} are certainly better.

### 9.7.5.1 Finding symbols to declare

The macro \texttt{analysing-special-variables-usage} can be used to find symbols that may be proclaimed global, which can improve performance. \texttt{analysing-special-variables-usage} also helps to identify inconsistencies in the code.

### 9.7.5.2 Coalesce multiple special bindings

If a set of specials are always bound at the same time, it is better to store the values in a single structure object and bind one special variable to that object, to reduce the overall number of special bindings.

### 9.7.6 Stack allocation of objects with dynamic extent

\texttt{(declare dynamic-extent)} will optimize these calls so that they allocate in the stack, in all cases:

- \texttt{&rest} lists
- \texttt{flet} functions and \texttt{labels} functions
- \texttt{(cons x y)}
- \texttt{(list ...)}
- \texttt{(list* ...)}
- \texttt{(copy-list x)}
- \texttt{(make-list x)}
- \texttt{(vector ...)}
(declare dynamic-extent) will also optimize these specific calls:

- (make-array n)
- (make-array n :initial-element x) without any other arguments
- (make-foo ...) where make-foo is an inline structure constructor. The default constructor is declared inline automatically when none of the defstruct slot initforms are calls to functions.
- (make-string n :element-type 'base-char)
- (system:make-typed-aref-vector n)

9.7.7 Inlining foreign slot access

Given a structure definition

(fli:define-c-struct foo-struct
   (a :int)
   (b :int))

you can inline access to a slot by declaring fli:foreign-slot-value inline and supplying the object-type:

(defun foo-a (struct)
  (declare (inline fli:foreign-slot-value))
  (fli:foreign-slot-value struct 'a :object-type 'foo-struct))

9.7.8 Built-in optimization of remove-duplicates and delete-duplicates

LispWorks optimizes cl:remove-duplicates and cl:delete-duplicates for lists when the test or test-not is one of a small set of known functions. These functions are currently cl:eq, cl:eql, cl:equal, cl:equalp, cl:=, cl:string=, cl:string-equal, cl:char= and cl:char-equal.

9.8 Compiler parameters affecting LispWorks

There are six compiler parameters that control the generation of information used by various LispWorks utilities, such as the debugger, and also by various editor commands, such as Show Paths From. By default, these parameters are all t, which allows you to use all the features of these utilities, at the expense of increasing compilation times.
These variables are initially set to \texttt{t} (in the LispWorks file \texttt{config/a-dot-lispworks.lisp}). To speed up compilation times, you should set these variables to \texttt{nil}. The variables can be controlled as a group by using the function \texttt{toggle-source-debugging}.

It is also possible to compile your code with counters or flags such that you can see which parts of your program have actually executed at run time, as described in Chapter 10, “Code Coverage”.
9 The Compiler
10

Code Coverage

Code Coverage in LispWorks allows you to compile your code with code execution counters, which then record when each piece of code is executed, and then display which parts of the program were executed and how frequently. Alternatively you can compile your code with a binary flag to record simply whether each piece was executed or not.

10.1 Using Code Coverage

Using Code Coverage involves four steps, described in this section:

1. Compiling the code to record code coverage information.
2. Loading the code.
3. Exercising the code.
4. Displaying the results.

Optionally, you can get a copy of the results and manipulate these before displaying them, as described in “Manipulating code coverage data” on page 121.
10 Code Coverage

10.1.1 Compiling the code to record code coverage information

Switch on generation of Code Coverage by either calling `generate-code-coverage` (which switches it globally), or using the macro `with-code-coverage-generation` (which switches it on only within the dynamic scope of the macro). Then compile your file(s) by calling `compile-file`. Alternatively you can use something that calls `compile-file` such as `compile-system`, menu command File > Compile... or the editor command Compile File.

Code Coverage works only when compiling into binary files, rather than into memory (which is what some editor commands such as Compile Buffer do). When `compile-file` is called with code coverage generation, it generates code that keeps track of execution and contains some extra data. This results in slightly slower code and larger binary files that use more memory when loaded.

10.1.2 Loading the code

Load your compiled files as usual by calling `cl:load`. Alternatively you can use something that calls `cl:load` such as `load-system`, menu command File > Load... or the editor command Load File.

When a file that was compiled with code coverage generation is loaded, it automatically adds itself to the internal `code-coverage-data` structure (overwriting existing data), and from that point any access to this structure (see below) will include information about the code in this file. Executing code that was compiled with code coverage generation always updates the internal `code-coverage-data` structure (it is not switchable).

10.1.3 Exercising the code

Decide what you want to check, and run the entry points.

Code Coverage measures which parts of the program were executed, so you need to decide what you want to check and call the entry points. In a graphical application, you need to display the main window and interact with it.
10.1.4 Displaying the results

There are two ways to view the results:

- as HTML files in a web browser, or
- with the Editor and Code Coverage Browser tools in the LispWorks IDE.

HTML display is done by calling `code-coverage-data-generate-coloring-html`, which in general generates one HTML file per source file in the `code-coverage-data`, and also an index HTML file with hyperlinks to all of them. Editor display is done by the function `editor-color-code-coverage`, which takes the name of a source file and creates a new editor buffer with the source colored according to the code coverage. Both functions take various keywords to control what they actually do. By default, both of them use the internal `code-coverage-data` structure, but can also use a manipulated `code-coverage-data`. See `code-coverage-data-generate-coloring-html` and `editor-color-code-coverage` for full details.

10.2 Manipulating code coverage data

Optionally you can manipulate `code-coverage-data` before displaying it, using the functions described in this section.

The `code-coverage-data` structure contains information about some set of files. Except for the internal `code-coverage-data`, this information does not change. The internal `code-coverage-data` object is updated whenever code that was compiled with code coverage runs. Also, when a binary file that was compiled with code coverage is loaded it adds itself to the internal `code-coverage-data` (overwriting any existing data associated with that file). Most of the functions for manipulating `code-coverage-data` can operate on the internal `code-coverage-data` structure by passing `t` as the data argument. See the specific functions for details.

The interface to code coverage data allows you to:

- Copy `code-coverage-data`, save it into a file, and load a `code-coverage-data` from a file.
These functions are `copy-code-coverage-data`, `copy-current-code-coverage`, `merge-code-coverage-data`, `filter-code-coverage-data`, `save-code-coverage-data`, `save-current-code-coverage` and `load-code-coverage-data`.

- Add or subtract two `code-coverage-data` structures. This means add or subtract all the corresponding counters from the two structures. This is allowed only if all the files that are in both structures are from the same compilation.

  For example, you may subtract the data of one test from another to see how they differ in the way they use their code.

  These functions are `add-code-coverage-data`, `destructive-add-code-coverage-data`, `subtract-code-coverage-data`, `destructive-subtract-code-coverage-data`, `reverse-subtract-code-coverage-data` and `destructive-reverse-subtract-code-coverage-data`.

- Clear the internal `code-coverage-data` (this means eliminating the files from it) or reset it (this means setting the counters to 0), or setting to another `code-coverage-data`.

  These functions are `clear-code-coverage`, `reset-code-coverage` and `restore-code-coverage-data`.

- Set an internal snapshot, and later compare with it.

  These functions are `set-code-coverage-snapshot`, `get-code-coverage-delta` and `reset-code-coverage-snapshot`.

- Generate statistics about the contents of the code coverage, by `code-coverage-data-generate-statistics`.

### 10.3 Preventing code generation for some forms

You can use the macros `error-situation-forms` and `without-code-coverage` to prevent generation of code coverage inside their body. For example, explicit calls to `cl:error` that are not expected to happen can be marked not to be counted. The system uses these in macros that call `cl:error` such as `cl:etypecase` and `cl:assert`. 
10.4 Code coverage and multithreading

By default, code compiled with code coverage uses non-atomic counters, which means that if the code runs in multiple threads it will occasionally drop a count. As the count is mostly used as heuristics this is usually not a problem (it will never drop all the counts, so you will not get 0 counts when there should be more than 0).

To record an exact count you can compile with atomic-p t (see generate-code-coverage and with-code-coverage-generation). Atomic incrementing may make the program run much more slowly, which is the reason that it is not the default behavior.

The code-coverage-data manipulation functions are thread-safe, and will not corrupt data or cause errors when running in parallel. However they are not atomic, so modifying the same structure in parallel will create inconsistent data. Reading the internal data while code with counters is executing may also generate an inconsistent data.

10.5 Counting overflow

The code coverage counters are 32-bit signed values (signed to allow negative values, which you can get when subtracting). Long tests can overflow in their frequently-called functions. That means that for these functions the counter is not that useful anymore. Also, you can end up on exactly 0, which looks as if the code was not executed. For heuristics that seems not to be a problem.

To avoid these problems with counter overflow, you can compile with a binary flag (initial value 0) instead of the counter. The flag switches to 1 when the code is called (see generate-code-coverage and with-code-coverage-generation). This loses the counting, but also generates smaller and faster code and uses less memory.

10.6 Memory usage and code speed

Collecting code coverage information makes the code larger and slower, but still workable. Compiling with binary flags results in code that is faster and smaller than code compiled with counters (see generate-code-coverage and with-code-coverage-generation) and it also reduces the size of the
data that code-coverage-data needs to keep. On the other hand you lose the counters, but if you do not need the counters it may be useful.

10.7 Understanding the code coverage output

In general there are five colors that are used to color the source code. Below these are named by the keyword argument to code-coverage-set-html-background-colors that changes them for the HTML coloring, and the default color is shown in parentheses:

- **fully-covered** (Green)
  - Forms where every part was covered, that is all of the source subforms were executed.

- **partially-covered** (GreenYellow)
  - Forms that are partially covered in a visible way, that is some of the source subforms were not executed.

- **hidden-partial** (Orange)
  - Forms that are partially covered in a non-visible way, that is all the source subforms were executed, but some of them were expanded to forms that were not completely executed.

- **uncovered** (Pink)
  - Uncovered forms, that is forms that were never executed.

- **eliminated** (DeepPink)
  - Forms that were completely eliminated by the compiler.

Note that by default only hidden-partial, uncovered and eliminated are shown, so all colored forms indicate something that was not covered. However, inside a hidden-partial form the fully-covered subforms are always colored, regardless of the setting of fully-covered.

When counters are displayed, they have their own background color counters (MediumAquamarine), except for negative counters that use color counters-negative (Gold). In addition the colors error (Red) and warn (Yellow) are used when adding error or warning messages.
10.7 Understanding the code coverage output

10.7.1 Eliminated forms
The compiler eliminates forms that it determines are not needed. These include:

- Forms that have no side effects and whose result is not used.
- Forms that will never be reached.

To deduce that code does not have side effects, the compiler needs to know that the function calls in it have no side effects. The only function calls that the compiler knows to be free of side effects are either system functions or automatically defined functions such as structure accessors.

Unreachable code can happen explicitly but it can also happen implicitly, when the compiler eliminates it. The commonest case is when the compiler uses type inference to infer what a predicate will return. For example, suppose you have this definition:

```lisp
(defun my-func (arg)
  (let* ((sum (+ arg 10)))
    (if (consp sum)
        (car sum)
        (* sum 7)))
)
```

Since `sum` is a result of the call to `+` it must be a number and therefore cannot be a cons. Hence the compiler can infer that the `consp` call will always return `nil`, replace the call to `consp` by `nil`, eliminate the call to `car`, and go straight to the `*` call. In the coloring, the call to `car` will be shown as eliminated, while the `if` form will be shown as fully-covered. That is a little counter-intuitive, because the `if` and the `consp` are not actually in the compiled code, but they were effectively evaluated by the compiler at compile time.

10.7.2 Displaying counters
By default, code coloring adds counters, indicating how many times a point in the code has been executed. Note that counts may also be negative, when the code coverage data that is displayed is the difference between datas (as generated by functions like `subtract-code-coverage-data`). When a count is 0 the counter is not displayed.

By default, if the counter for a subform is the same as the counter of the parent form, the counter for that subform is not displayed. Thus when you see a sub-
form without a counter, it means it was executed the same number of times as its parent. The counter is displayed even if it is the same in situations when the code coverage status of the forms is different, for example if the parent is partially-covered but the subform is fully-covered.

It is possible to force all counters to be displayed, by passing :show-counters :all to code-coverage-data-generate-coloring-html and the other displaying functions. Note that the compiler does not generate counters for forms that it can deduce will be always executed the same number of times as the parent, so for these it will never display counters.

10.7.3 Function forms where the function is not actually called

Entering a function form does not necessarily call the function, because during evaluation of the arguments there may be an exit out of the form. This exit can be local or non-local, and in general the compiler cannot tell whether it will happen or not, though it can be sure that it will not happen for specific cases like local variable references and self-evaluating forms. Therefore the compiler adds a counter just before the function is called. It is not obvious how to display this counter, and when it is 0 what part of the form has not been executed.

The current coloring regards the function name as the part of the form that is "being evaluated" when the function is actually called. Therefore it inserts the counter before the function name, and if it is 0 it colors the function name in the uncovered color. That means that in some cases you have a counter before the opening parenthesis that counts the times that the form was entered, and a counter after the opening parenthesis that counts the times that the function was called. Since the coloring normally skips counters that are inside a form where they have the same count as the counter of the form, you will see both numbers only when they are different.

10.7.4 Partially hidden

Partially hidden forms are forms where all the source code as shown was executed, but some macro expansion of it contains a conditional where one of the branches was not executed. Thus there is not any specific form to color as uncovered, but some part of the actual code was not executed. These forms are colored by a special color hidden-partial, which is Orange by default.
For example, if you have this code:

```lisp
(defmacro did-it-p ()
  '(if *done* nil t))

(defun go-next-p ()
  (did-it-p))
```

and whenever `go-next-p` is called `*done*` is true, then the false branch of the `if` form was not executed, but inside the definition of `go-next-p` there is no source that was not executed. Thus the form `(did-it-p)` is partially hidden.

Partially hidden forms can have subforms that are fully covered. These forms are colored by the color `fully-covered` even if fully covered forms are not covered otherwise (the default behavior), to clarify that they were fully covered.

### 10.8 Coloring code that has changed

The code coverage data does not keep the source code. Instead, it keeps a reference to the code that the reader saw (when called from `compile-file`). When it adds colors, it re-reads the source file. That means it needs the original source file for coloring. If the source file was modified, it adds a warning in the beginning of the file, but tries to color it anyway.

If what the reader sees has not changed (that is the only changes involve only comments and whitespace) the coloring will work properly. Changes to what the reader sees, however, will confuse the coloring. In general, subforms that are modified are miscolored, but code outside the modified subform colors properly. For a top level form, if you modify it, this form will not color properly, but all the other forms will color properly. If you remove or add a top level form, all the following forms will not color properly. Note that this applies even if you do something like adding (or removing) a `progn` around some forms, which although it does not affect the compiler does cause the reader to see different forms.

Thus if you modify your code, the coloring becomes less reliable. In most cases this is not a big problem, but in many cases it is probably better to copy your source tree and compile the copy with code coverage, so you can continue to modify the source while reviewing the code coverage output.
This chapter introduces some basic ideas of memory management, and then discusses the LispWorks memory management system in more detail. The chapter also introduces the functions and macros needed to control memory management. Full details of all the symbols mentioned here are given in Chapter 38, “The HCL Package” and Chapter 49, “The SYSTEM Package”.

11.1 Introduction

Automatic memory management is one of the most significant features of a Lisp system. Whenever an object, such as a cons cell, is required to hold an aggregate of values, the system calls the appropriate function to create a new object and fill it with the intended values. The programmer need not be concerned with the low level allocation and management of memory as the Lisp system provides this functionality automatically.

When an object is no longer required (that is, it has become “garbage”), the system must automatically reclaim (“collect”) the space it occupies and reallocate the space to a new object. Whenever the space for new objects is exhausted, a “garbage collector” (GC) is run to determine (by a process of elimination) all the existing objects that are still required by the running program. Any other objects still in the image are necessarily garbage, and the space they occupy can be reclaimed.
11 Memory Management

For a description of how LispWorks uses the address space of different Operating Systems, and factors affecting the maximum image size, see “Address Space and Image Size” on page 458.

Garbage collection with a naive algorithm is extremely inefficient.

The LispWorks GC works in unison with the memory allocator to arrange allocated objects in a series of “generations”. Each generation contains objects of a particular age. In practice most Lisp data objects are only required for a very short period of time. That is, they are ephemeral. The LispWorks GC concentrates its efforts on repeatedly scanning the most recent generation. Such a scan requires only a fraction of a second and reclaims most of the space allocated since the last collection. Any object in the most recent generation that survives a number of such collections is promoted to the next youngest generation. Eventually this older generation becomes full, and only then is it collected. The generations are numbered from 0 upwards, so that generation 0 is the youngest.

The remainder of this chapter describes the LispWorks GC in more detail. The implementation and the programmatic interface differ between 32-bit and 64-bit LispWorks.

11.2 Guidance for control of the memory management system

11.2.1 General guidance

The memory management is designed with the intention that the programmer will have to do very little or nothing about it. In general, we believe that the design is quite successful, and in most cases you do not have to do anything. The main exception to this is dealing with long-lived data in long-lived processes in 32-bit LispWorks.

Before doing anything about memory management, you should be familiar with the function `room`, and use it frequently. There is no point at all in trying to tune the memory management without knowing the sizes of your application, as output by `room`.
11.2 Guidance for control of the memory management system

The data and code in the LispWorks image can be categorized according to how long they live, as follows:

1. Short-lived data
2. Long-lived data
3. Permanent data

Note that the distinction is not in the data itself, but in the existence of pointers to it.

In general, you rarely need to worry about short-lived data, and have to worry about permanent data only if you have a large amount of it. In short-lived applications you do not need to worry about long-lived data either, so there is a good chance that you do not have to worry about memory management at all.

In long-lived applications, you certainly need to consider long-lived data in 32-bit LispWorks, and maybe in 64-bit LispWorks.

11.2.2 Short-lived data

Normally you should not do anything about handling of short-lived data, because the default settings are good enough for almost all situations. Sometimes you may hit a situation where the settings are not good ("pathological case"). However, it would normally require a deep understanding of the memory management system to deal with such a situation, and we will in general consider this as a bug and try to fix it. Therefore if you find such situation you should report it to Lisp Support, following the guidelines at www.lispworks.com/support/bug-report.html.

Problems with short-lived data normally just reduce the performance of some part of your application. Normally the best solution is to optimize the code to do less work, including allocating less.

To do that, first find the bottlenecks in your application by using profile (and start-profiling and stop-profiling). time and extended-time can then be used to determine how long specific operations take, how much they allocate and, for long operations, how long they spend in garbage collection. Use this information to decide what to try to optimize.
11.2.3 Long-lived data

Long-lived data is data that lives long enough to be promoted to the highest generation to which promotion occurs automatically (the "blocking generation"), but later becomes garbage. The blocking generation is 2 in 32-bit LispWorks and (by default) 3 in 64-bit LispWorks.

You can check which generation individual objects are in (by generation-number), but normally you want to know the total amount of data in various generations. The function room is used for that. In general, it is useful to call

(room)

and sometimes also

(room t)

periodically (every 5 minutes) and log the output. In servers, such logs are essential. From this output you can see how the sizes of the various generations change over time.

If the output shows that the blocking generation grows too much, even though permanent data is not added, you will need to do something about it. In 64-bit LispWorks there is a good chance that you do not have to do anything. In 32-bit LispWorks long-lived processes (for example servers) probably need to do something.

The main thing you will do is calling (gc-generation t). This garbage collects the blocking generation. You should check the state of the memory after calling it by calling room again. If the amount of allocated data (as opposed to total size) did not reduce, you may have a memory leak that causes accumulation of permanent data that does not die.

If gc-generation does free data (that is, the allocation reduces significantly), you probably need to add calls to it to your application.

Compatibility note: In 32-bit LispWorks version 5.1 and earlier, the documented way to collect generation 2 is to call (mark-and-sweep 2). (gc-generation t) does what (mark-and-sweep 2) does, plus some additional operations that improve the performance of allocation. It also has the advantage that it is the same call that is used in 64-bit LispWorks. We recommend always using gc-generation.
To decide when to call `gc-generation`, you need to consider the following:

1. You need to prevent excessive growth of the process.
2. You want to avoid calling `gc-generation` when the application needs to respond quickly.
3. The call will be more effective if it is done between chunks of work than in the middle of a chunk of work.

We now discuss these considerations in detail:

1. You can follow the overall size of the process by looking at the output of `(room nil)`, or programmatically by using the result of `room-values`. The definition of "excessive growth" depends on the machine that you are running on and what the server actually does. Normally you want to avoid the need for paging, so you should try to keep the size of the image below the size of real memory that it can use. For 32-bit LispWorks on modern machines that have a lot of memory, the limit will be the amount of address space the machine has. In addition, garbage collecting a larger image takes more time. In a typical 32-bit application, 100-200 MB would be the target, though it can be larger. In a 64-bit application the limit is the size of the real memory.

2. `(gc-generation t)` can take a significant amount of time. 32-bit LispWorks on a modern machine can collect 100-200 MB in less than a second if it does not page. If it pages, or has a slower CPU, it takes more time. The 64-bit GC is generally faster and better, as long as it does not page, but since you normally deal with much more data in 64-bit images, there may be significant delays in 64-bit LispWorks. If such delays are a problem for your application, you should try to call `gc-generation` at times when it is less of a problem. Use `time` to find out how long `gc-generation` takes in various situations.

3. If you can identify places where there are no active chunks of work, you can try to place calls to `gc-generation` in these places. For servers, this is likely to be much less important than the two considerations above, but for an application that computes results using large amounts of data, this may be a significant consideration.

In 32-bit LispWorks, by default, generation 2 (which is the "blocking generation") is not collected automatically, because such collection may take a
significant amount of time, so most programmers need to control when it actually happens. You can change this by using `collect-generation-2`, but usually you need better control, and do a collection of generation 2 when it is appropriate. Therefore if your application generates long-lived data, you need to add calls to `gc-generation`.

Even if you find that your application does not generate long-lived data (that is, generation 2 does not grow), it is probably a good idea to keep checking, in case some circumstances do cause it to generate long-lived data.

In 64-bit LispWorks by default generation 3 (the "blocking generation") is collected automatically, so there is a good chance that you do not have to do anything. However, you may want to call `gc-generation` explicitly when you know it is a good time to do it. You may also want to block automatic calls if they take too long: use `set-blocking-gen-num` to do that. If generation 3 becomes very big (Gigabytes), you may also consider using `marking-gc` instead of `gc-generation`.

Once you set up `gc-generation` calls, you may still see the image growing even though the allocation does not grow that much. That is normally the result of fragmentation. In 32-bit LispWorks you can use `check-fragmentation` to check for fragmentation, and `try-move-in-generation` to prevent it if needed. See “Controlling Fragmentation” on page 144 for a discussion.

In 64-bit LispWorks you have a problem with fragmentation only if you use `marking-gc`. `marking-gc` has keyword arguments that can be used to reduce fragmentation, and there is a good chance that using these will be enough to avoid serious fragmentation. `gc-generation` can be used occasionally to eliminate all fragmentation. Check for fragmentation by using `gen-num-segments-fragmentation-state`.

### 11.2.4 Permanent data

Permanently-living data will typically be the actual code of the application, and maybe also data that never goes away.

Because the data never goes away, it is best to put it outside normal garbage collection, which means promoting it to the highest generation. This is done by `clean-down`, which is called automatically (by default) when saving an image (whether by `save-image` or `deliver`). In most cases that is the right
time to do it, so normally you do not need to call `clean-down` explicitly. In some situations you may want to call it yourself, and sometimes you want to avoid the call when saving an image with a lot of non-permanent data. To control the automatic call, see `save-image` and `deliver`.

There are several things that need to be considered when using `clean-down`:

1. If the permanent data is only a small amount compared to the long-lived data, it is not obvious that `clean-down` is needed, specially if you use a saved (or delivered) image where the code and maybe some data was already promoted.

2. `clean-down` promotes all the data that is live (that is, pointed to from some other live object) in the image when it is called. If the image contains data that is live, but later becomes garbage, it will be promoted and hence not collected until another call to `clean-down`, which will make the image unnecessarily larger. Since this data is not being accessed, the effect on performance is small, but if there is a lot of it the effect may be significant.

3. `clean-down` needs extra memory to operate, especially in 64-bit LispWorks. For very large 64-bit LispWorks images `clean-down` may fail due to running out of swap memory.

4. `clean-down` takes a significant amount of time. If it does not cause paging, it should take seconds, but if it needs to page it may take much longer. You therefore should avoid calling it when you need the application to respond reasonably quickly.

### 11.3 Memory Management in 32-bit LispWorks

This section describes the garbage collector (GC) in 32-bit LispWorks 7.1.

In LispWorks for UNIX and LispWorks for Macintosh, the implementation is not significantly different to that in LispWorks 4.x, LispWorks 5.x or LispWorks 6.x.

In LispWorks for Windows and LispWorks for Linux, the implementation has changed since LispWorks 4.x and you may notice performance improvements relative to those versions.
11.3.1 Generations

In memory, a generation consists of a chain of segments. Each segment is a contiguous block of memory, beginning with a header and followed by the allocation area.

The first generation normally consists of two segments: the first segment is relatively small, and is where most of the allocation takes place. The second segment is called the big-chunk area, and is used for allocating large objects and when overflow occurs (see below for a discussion of overflow).

The second generation (generation 1) is an intermediate generation, for objects that have been promoted from generation 0 (typically for objects that live for some minutes).

Long-lived objects are eventually promoted to generation 2. Note that generation 2 is not scanned automatically. Therefore these objects will not be reclaimed (even if they are not referenced) until an explicit call to a GC function (for example `gc-generation` on `t`, or `clean-down`) or when the image is saved. Normally, objects are not promoted from generation 2 to generation 3, except when the image is saved.

Generation 3 normally contains only objects that existed at startup time, that is those that were saved in the image. Normally it is not scanned at all, except when an image is saved.

Note that the division between the generations is a result of the promotion mechanism, and is not a property of a piece of code itself. A piece of system software code that is loaded in the system (for example, a patch) is treated the same as any other code. The garbage collection code is explicitly loaded in the static area using the function `switch-static-allocation`.

11.3.2 Allocation of objects

Normal allocation is done from a buffer, called the small objects buffer. The GC maintains a pointer to the beginning and end of the buffer, and allocates from it by moving one of the boundaries. When the buffer becomes too small the GC finds another free block and makes that the buffer.

In non-SMP LispWorks there is only one global small objects buffer. In SMP LispWorks, each process may have its own “local” small objects buffer (in
addition to the global one). The system decides dynamically which process should have a local buffer and which not. In general processes that do any significant amount of work have a local buffer, and most of their allocation would be from local buffers.

When there is an overflow the small object buffer is allocated in the big-chunk area, and then a bigger buffer is allocated (see below).

### 11.3.2.1 Allocation of static objects

Objects that cannot be moved are allocated in special segments, called static segments. These can be in any generation, but are in generation 2 by default. Such objects include:

- Code that must not move during garbage collection, in particular the code and data of the GC itself
- Arrays created by `make-array` with `allocation : static`. This is the preferred way to allocate a static array.
- Objects allocated explicitly in the static area, by `in-static-area` or by use of `switch-static-allocation`.
- Foreign code loaded from a non-shared library via `link-load:read-foreign-modules`. This applies to LispWorks for SPARC Solaris only.

Because static objects are not allowed to move, the static segments are not allowed to move. This implies that if there is a static segment in a high address the image size cannot be reduced below this size. Applications that use a lot of static area normally allocate additional static segments, and thus grow without being able to shrink again. This can be prevented by enlarging the initial static segment, which is in a low address. Use the function `enlarge-static` to increase the size of the initial static segment. (Use `(room t)` to find its current size.)

### 11.3.2.2 Allocation in different generations

Objects that are known to have long life can be allocated directly in a higher generation, by using `allocation-in-gen-num` and `set-default-generation`. Note that both these functions have a global effect, that is any object
allocated after a call to `set-default-generation` or within the body of `allocation-in-gen-num` is allocated in the specified generation, unless it is explicitly allocated in a different generation. Therefore careless use of these functions may lead to allocation of ephemeral garbage in high generations, which is very inefficient. Conversely, if a long-lasting object is allocated to a low generation, it has to survive several garbage collections before being automatically promoted to the next generation.

The best way to control the allocation generation for an array is to call `make-array` with `allocation :long-lived` or a number.

See also “Allocation of interned symbols and packages” on page 164 and “Allocation of stacks” on page 164.

### 11.3.3 GC operations

Mark and sweep is the basic operation of reclaiming memory, and it is done in two stages:

- **Mark**: All objects that are alive in the generation being garbage collected and in younger generations are marked as alive. (Alive means pointed to by some other live object.)

- **Sweep**: All unmarked objects in the generations being garbage collected are added to the free blocks, and all marked objects are unmarked.

A mark and sweep operation is always on all the generations from 0 to a specific number.

A mark and sweep operation can be caused explicitly by calling `gc-generation`.

Promotion is the process of moving objects from one generation to the next generation. An object is marked for promotion after surviving a specific number of mark and sweep operations, but may be promoted before that. The number of survivals is specific to each segment.

Promotion does not free objects.
11.3.4 Garbage collection strategy

When the GC runs out of memory, it has to find more memory. Normally (that is, when allocating in generation 0) the first operation is a mark and sweep. Before performing the mark and sweep, the GC compares the amount of memory allocated since the previous mark and sweep with the `minimum-for-sweep` value, which is set by `set-gc-parameters`. If the amount allocated is less than `minimum-for-sweep` the GC does not do a mark and sweep, but causes an overflow (described below). This prevents an excessive number of mark and sweep operations in periods when the program allocates a large amount of data which stays alive.

If more than `minimum-for-sweep` has been allocated, a mark and sweep operation takes place. After this operation the GC checks that the segment it was trying to allocate to has more free space than the minimum free space for this segment. If the remaining free space is less than `minimum-free-space`, the GC tries to create more free space by promoting objects from the segment.

Before promoting, the GC performs two checks. First, it checks that there are enough objects marked for promotion to justify a promotion operation. The minimum free space for a segment is set by `set-minimum-free-space`, and can be shown by `(room t)`.

Second, the GC checks that there is enough free space in the next generation to accommodate the promoted objects. If there is insufficient space, the GC tries to free some, either by a mark and sweep on the next generation, promoting the next generation, or by enlarging the generation.

The minimum amount of space for promotion is the value `minimum-for-promote`, which is set by `set-gc-parameters`.

If there is insufficient space, and there are not enough objects marked for promotion, the GC increases the size of the image, by overflow, as described below.

On Motif only, note that the GC monitor window does not indicate a mark and sweep of generation 0, as this operation takes a small amount of time (it would take longer to change the display of the window). The GC monitor window appears only in the Motif IDE.
11.3.5 Memory layout

11.3.5.1 Linux

On Linux, the default initial heap is mapped at address \#x20000000 (0.5 GB). LispWorks then tries to locate the location of dynamic libraries, and marks a region around these libraries that should not be used (by default 64 MB from the bottom). In most cases this suffices to avoid clashes.

Problems can arise if the memory at \#x20000000 or above is already used by another part of the software. If that memory gets used before LispWorks is mapped, LispWorks must be relocated elsewhere, typically to a higher address, as described in “Startup relocation of 32-bit LispWorks” on page 460.

If the memory above LispWorks gets used by other parts of the software after LispWorks was mapped, it may be possible to avoid the problem by reserving some memory above LispWorks by supplying ReserveSize.

The location of dynamic libraries differs between Linux configurations, and that needs to be taken into account. For most cases, including the cases where the libraries are mapped at \#x40000000 or somewhere above \#x28000000, the mechanism for detecting libraries works and no action is required.

In principle LispWorks (32-bit) for Linux can grow up to some distance below \#xBF000000 (almost 2.5 GB), though this depends on the OS kernel allowing this size.

Note: In LispWorks 5.0 and previous, we told some customers to relocate above the libraries, for example at \#x50000000 or \#x48000000, but this should not be needed in LispWorks 7.1.

11.3.5.2 FreeBSD

By default, LispWorks is mapped at \#x30000000.

Problems may arise if something uses memory above \#x30000000. If this memory is used before LispWorks is mapped, LispWorks must be relocated elsewhere, typically to a higher address, as described in “Startup relocation of 32-bit LispWorks” on page 460.
If the memory above LispWorks gets used by other parts of the software after LispWorks was mapped, it may be possible to avoid the problem by reserving some memory above LispWorks by using ReserveSize.

Normally the dynamic libraries are mapped at #x28000000, and therefore LispWorks can grow without a problem.

In principle LispWorks can grow up to some distance below #xC0000000 (almost 2.25 GB), though this depends on the OS kernel allowing this size and how many threads you have running.

11.3.5.3 x86/x64 Solaris

The default initial heap is mapped at address #x10000000 (0.25 GB). LispWorks then tries to locate the location of dynamic libraries, and marks a region around these libraries that should not be used (by default 64 MB from the bottom). In most cases this suffices to avoid clashes.

Problems can arise if the memory at #x10000000 or above is already used by another part of the software. If that memory gets used before LispWorks is mapped, LispWorks must be relocated elsewhere, typically to a higher address, as described in “Startup relocation of 32-bit LispWorks” on page 460.

If the memory above LispWorks gets used by other parts of the software after LispWorks was mapped, it may be possible to avoid the problem by reserving some memory above LispWorks by supplying ReserveSize.

11.3.5.4 Windows and Macintosh

LispWorks (32-bit) for Windows and LispWorks (32-bit) for Macintosh both map by default at #x20000000. Since these platforms support reservation, normally you will not need to do anything special about this.

Problems may however arise if LispWorks operates in conjunction with non-relocatable software which insists on using addresses at #x20000000 or some distance above, in which case you will need to relocate LispWorks, as described in “Startup relocation of 32-bit LispWorks” on page 460.

LispWorks (32-bit) for Windows can in principle grow up to some distance below #x80000000 (almost 1.5 GB) but there is always the possibility that
some DLL will be mapped in this region. On startup, it reserves 0.5 GB above its location, so that much is guaranteed.

LispWorks (32-bit) for Macintosh can grow to around 2.7 GB. You can relocate it only on the Intel architecture.

11.3.5.5 AIX

The image is mapped at #x31000000.

The 16 MB (= #x1000000) above #x30000000 allows it to grow down a little, which is used for static objects. In most cases it does not use this memory. The heap can grow up from #x31000000 to #xd0000000, where the libraries are mapped.

LispWorks also maps some memory at #xe0000000, which is a read-only area containing code and some strings.

Foreign data (malloc and stacks) is in the normal AIX place, the 256 MB between #x20000000 and #x30000000.

The heap is relocatable as described in “Startup relocation of 32-bit LispWorks” on page 460. However, to actually allow a larger area for foreign data, you also need to enable large address-space as far as AIX is concerned. This can done either at the time LispWorks is invoked by the environment variable LDR_CNTRL with maxdata, or statically on the LispWorks executable by using ldedit with -bmaxdata. For details see the entry for ”Large program Support” in the AIX documentation (search for ”Large program Support” at www.ibm.com).

When relocating it generally advisable to leave several megabytes between the heap and the top of the foreign data, in case LispWorks needs more static area. However, this is not essential, because in most of cases LispWorks will not need it, and it can allocate static areas higher.

11.3.6 Approaching the memory limit

If your program allocates a lot you may reach the limit of memory that LispWorks can use. The limit depends on the architecture as described in “Memory layout” on page 140.
When LispWorks actually reaches the limit it will fail to communicate with the user due to allocation errors. To avoid this situation, LispWorks informs the user earlier that it is approaching the limit of memory. It first checks whether you set the approaching memory callback (by set-approaching-memory-limit-callback), and if there is a callback calls it. If there is no callback or the callback returns, LispWorks signals an error of type approaching-memory-limit (which is a subclass of cl:storage-condition).

The function memory-growth-margin can be used to see how much LispWorks "believes" that it can grow.

The callback can be used to effectively ignore the condition, but this is a bad idea in general, because it will probably lead to an error later when LispWorks actually reaches the limit, and then it may crash in a bad way. To be safe, the callback should either cleanup and exit, or free a substantial amount of memory. You can reasonably continue only if a crash is not going to cause a serious damage.

### 11.3.7 Overflow

If the amount allocated from the previous mark and sweep operation is less than :minimum-for-sweep, the GC does not perform a mark and sweep. Instead it allocates a small-objects buffer in the big-chunk area (the second segment in the first generation). The minimum and maximum sizes of this buffer are specified by :minimum-overflow and :maximum-overflow, which can be set by set-gc-parameters. If the GC fails to find a buffer of this size, it looks for a smaller buffer, and if that fails it enlarges the big-chunk area (and the process size) by the amount needed to allocate a buffer of the size of the currently allocated area in generation 0, up to a maximum amount specified by :maximum-overflow.

### 11.3.8 Behavior of generation 1

When objects are promoted from generation 0 to 1, and there is not enough space in generation 1, the GC tries to free space in generation 1. The first step is to check whether sufficient space can be freed by promoting the objects marked for promotion. If this is the case the GC promotes these objects from generation 1 to generation 2. (In practice, this rarely happens.) If this check fails the GC marks and sweeps generation 1. If not enough space is freed by
this mark and sweep, than either all the objects in generation 1 are promoted, or generation 1 is expanded. This is controlled by \texttt{expand-generation-1}, which specifies whether expansion or promotion takes place.

If generation 1 is expanded, the amount it tries to expand by is the value \texttt{:new-generation-size} (set by \texttt{set-gc-parameters}) in words (that is, multiples of 4 bytes), or the amount of free space needed, whichever is bigger. If \texttt{:new-generation-size} is 0, it is not expanded. In this case part of the objects marked for promotion are not promoted.

\subsection*{11.3.9 Behavior of generation 2}

Normally generation 2 is not garbage collected. If the system runs out of space in this generation, it expands it, using the value of \texttt{:new-generation-size} multiplied by two. Garbage collection of generation 2 can be caused by calling the function \texttt{collect-generation-2} with appropriate argument.

\subsection*{11.3.10 Forcing expansion}

If you know that a given generation will need to grow, you can save the GC the work by calling \texttt{enlarge-generation} to expand the generation in advance.

\subsection*{11.3.11 Controlling Fragmentation}

Some applications periodically free (that is, stop using) a substantial amount of data that lived for long enough to reach generation 2 (use \texttt{room} or \texttt{room-values} and \texttt{generation-number} to follow the behavior of objects). In this case, \texttt{gc-generation} should be called on generation 2, to collect these data and re-use the memory. Repeated cycles like this may cause fragmentation, which will slow down promotion into generation 2. This manifests itself in significant pauses, typically of a few seconds. \texttt{try-move-in-generation} or \texttt{try-compact-in-generation} can be used to reduce the fragmentation, and hence to reduce the pauses. Because these functions themselves take some time, they should be called when such a pause is acceptable.

'Moving' a segment means moving objects out of the segment to another segment, leaving the segment empty. This reduces the fragmentation in the generation, and it is normally much faster than compact. Therefore in almost all

\clearpage
cases, try-move-in-generation is better than try-compact-in-generation.

The actual decision to use these functions will be typically based on the results of check-fragmentation. For example, the following function checks whether there is more than 10 MB free area in generation 2 in blocks of 4096 bytes or larger (tlb, third return value of check-fragmentation). If there is not, and the free area in generation 2 (tf) is more than four times the free area in large blocks, it calls try-move-in-generation. Because try-move-in-generation gets a time-threshold of 0, it returns after moving at most one segment. (It will not move any segments if none of them looks fragmented.)

(defun call-memory-functions()
  (gc-generation t) ; first collect all dead objects
  (multiple-value-bind (tf tsb tlb)
      (check-fragmentation 2) ; check the fragmentation
        (when (and (> 10000000 tlb) ;
          (> (ash tf -2) tlb)))
          (try-move-in-generation 2 0))))

A function such as this can be called at times when a pause of a few seconds is acceptable, and it will keep the memory of generation 2 less fragmented.

It is not possible to give definitive guidance here on how to use try-move-in-generation or try-compact-in-generation, because it depends on the way the application uses memory. In general, these functions will always improve the behavior of the application. Therefore the main problem is to identify points in the execution of the application where they can be called without causing unacceptably long pauses.

11.3.12 Summary of garbage collection symbols

The remainder of this chapter summarizes which functions are useful in which circumstances. See also “Common Memory Management Features” on page 163. For full details of these functions, see their reference entries.

11.3.12.1 Determining memory usage

To determine memory usage (useful when benchmarking), use the functions room, total-allocation and find-object-size. The function room-values is suitable for programmatic use: it returns the values that room prints.
In 32-bit LispWorks, `memory-growth-margin` returns the amount by which the Lisp heap can grow, if `set-maximum-memory` has been called.

### 11.3.12.2 Allocating in specific generations

Arrays can be allocated static or in a higher generation using the `allocation` argument in `make-array`.

To control the allocation of other objects to generations, use `allocation-in-gen-num`, `get-default-generation`, `set-default-generation` and `*symbol-alloc-gen-num*`.

### 11.3.12.3 Controlling a specific generation

To control the behavior of a specific generation, use `clean-generation-0`, `collect-generation-2`, `collect-highest-generation`, `expand-generation-1` and `set-minimum-free-space`.

### 11.3.12.4 Controlling the garbage collector

The functions that are most likely to be useful for controlling the GC are `room`, `check-fragmentation`, `gc-generation` and `try-move-in-generation`.

Other potentially useful functions and macros are `avoid-gc`, `get-gc-parameters`, `gc-if-needed`, `enlarge-generation`, `normal-gc`, `set-gc-parameters`, `with-heavy-allocation` and `try-compact-in-generation`.

### 11.4 Memory Management in 64-bit LispWorks

This section describes the garbage collector (GC) in 64-bit LispWorks.

#### 11.4.1 General organization of memory

The memory in 64-bit LispWorks is arranged in segments, which belong to generations. Unlike 32-bit LispWorks, segments are sparsely allocated in memory, that is they are not contiguous.

Each segment has an allocation type, which defines the type of objects that the segment contains. The system creates and destroys segments as needed. A generation may or may not contain a segment for a specific allocation type,
and a generation may contain more than one segment for any particular allo-
cation type. Segments may change in size.

You can see the allocation for each allocation type in the output of:

```
(room)
```

Additionally you can see the segments of each generation in the output of:

```
(room t)
```

After the total allocation in each generation, this prints the allocation type for
each segment followed by the hexadecimal address range for allocating
objects.

You can also use

```
(room :full)
```

which does not produce segments information, but prints allocated amounts
by allocation types.

### 11.4.2 Segments and Allocation Types

Some GC interface functions take an allocation type as an argument, which is
one of the keywords below. There are two categories of allocation type.

The main allocation types, which can be used as the `what` argument to the
function `apply-with-allocation-in-gen-num`, are:

- `:cons` The segment contains only conses.
- `:symbol` The segment contains only symbols (and does not include symbol names or any of the other properties of symbols).
- `:function` The segment contains only function objects.
- `:non-pointer` The segment contains only objects that do not contain pointers (strings, specialized numeric arrays, double-floats).
11 Memory Management

:other

The segment contain other objects, that is any object that contain pointers, and is not a symbol, cons or a function.

The derived allocation types are:

:mixed

The segment contains a mixture of :other, :function and :symbol, but not :cons or :non-pointer.

:cons-static

The segment contains cons objects that are static.

:non-pointer-static

The segment contains objects that do not contain pointers and are static (currently stacks are also allocated in these segments).

:mixed-static

The segment contains a mixture like :mixed, but static.

:weak

The segment contains weak objects (arrays, and internals of weak hash tables).

:other-big

The segment contains a single very large simple vector. The vector is static.

:non-pointer-big

The segment contains a single very large non-pointer object (a string or a specialized numeric array). The vector is static.

Segments of allocation type :other-big or :non-pointer-big can be as large as required to hold their object.

For all other allocation types, the size of each single segment is restricted. The implementation limit is currently 256 MB, and you can specify a smaller limit using set-maximum-segment-size.

11.4.3 Garbage Collection Operations

In 64-bit LispWorks there are two methods of garbage collection: copy (the default for all non-static objects) and mark and sweep (also referred to simply as mark) for static objects and under user control.
The two methods can be mixed within the same garbage collection operation and generation, but a segment is collected using only one of mark or copy in a given operation.

When a segment is collected using the copying method, the objects within it can either be copied to another segment in the same generation or can be copied to a segment in a higher generation. The latter case is called promotion. The automatic garbage collection copies with promotion until the objects reach the blocking generation, which is collected in a specific way as described in “Generation Management” on page 149.

### 11.4.4 Generation Management

In general, higher generations contain objects that live longer and are therefore much less likely to die. Each garbage collection only collects the generations up to some number, and never reclaims the objects in higher generations.

Objects move between generations by being promoted. For most allocation types, this means that the GC copies the objects from a segment in one generation to a segment in a higher generation. For allocation types `:other-big` and `:non-pointer-big`, the objects are not actually copied when they are promoted; but instead the whole segment is re-attached to the higher generation. The automatic garbage collection promotes objects until they reach the blocking generation.

In the default configuration, there are 8 generations, numbered from 0 to 7. Generation 7 is used to keep objects that survived saving the image. Generations 4, 5 and 6 are not used. Generation 3 is the blocking generation, where long-lived objects accumulate. Generations 0, 1, and 2 are ephemeral, and objects that survive a garbage collection in each of these generations are promoted to the next generation.

### 11.4.5 Tuning the garbage collector

The GC settings are tuned for typical cases, so in general you do not need to change them. If you are considering tuning the GC, contact Lisp Support.

The main tools for seeing how the GC behaves are the macro `extended-time` and periodical calls to `room`.
In the output of `room` (or the more verbose `room t`), the allocation in each generation is presented according to the allocation type, which may be useful to decide on possible tuning.

`extended-time forms` outputs the time spent in garbage collection, whether automatic or called explicitly. The time is shown according to the maximum generation number that was collected and to whether it was a standard garbage collection (automatic and calls to `gc-generation`) or a marking garbage collection (calls to `marking-gc`).

In addition to `room` and `extended-time`, there are also the functions `count-gen-num-allocation`, `gen-num-segments-fragmentation-state`, and `set-automatic-gc-callback`. These functions can be used to collect information about automatic garbage collection operations.

The profiler can also help determine whether the settings can be improved for your application. See Chapter 12, “The Profiler” for details of that.

### 11.4.5.1 Interface for tuning the GC

The main interfaces are those which control the blocking generation.

For generations lower than the blocking generation, objects that survive are promoted, and the system does not automatically promote objects to higher generations. Thus if the application generates long-lived objects, they will accumulate in the blocking generation.

The behavior when the blocking generation grows is controlled by `set-blocking-gen-num` and `set-gen-num-gc-threshold`. It may also be useful to set the maximum segment size with `set-maximum-segment-size`.

Explicit garbage collection can be done by calling `gc-generation` and `marking-gc`. Since repeated use of `marking-gc` will cause a lot of fragmentation, the arguments `what-to-copy` and `max-size-to-copy` can be used to specify that part of the data should be collected by copying.

`gc-generation` can also be used to promote objects to a higher generation than the blocking generation.

It is normally less important to tune the ephemeral segments, that is the segments below the blocking generation. Functions that may be useful include
11.5 The Mobile GC

The Mobile GC is a 64-bit GC that is written to run on 64-bit iOS (in the future it may be used on other platforms, for example 64-bit Android). When LispWorks is delivered for 64-bit iOS, the "saved image" (the code in the object file that delivery creates) switches automatically to use the Mobile GC. Thus you are always using the Mobile GC when running on 64-bit iOS, and you are not required to do anything about it.

The default parameters of the Mobile GC are intended to be useful for most applications and in many cases you do not need to do anything to tune the Mobile GC.

11.5.1 Mobile GC changes to common functions and macros

This section describes changes to the behavior of GC-related functions and macros when using the Mobile GC compared to the ordinary 64-bit GC. For most applications, room, and maybe gc-generation, are the only interesting functions. Specific functions for the Mobile GC are not discussed in this section.

`mobile-gc-p`

Returns true when running the Mobile GC.

`room`

The output of room is different for the Mobile GC. The last line (and the entire output of (room nil)) is the same, but the more detailed output is different. Without any argument, or if the argument is :default, LispWorks outputs the allocated and free sizes according to these types:

- **Cons**: cons object only.
- **Other**: All other objects, except static objects and large objects (> 1 MB).
Static  
Static objects.

Large  
Large objects (> 1 MB). Note: this threshold may change in the future, but it is fixed in the current version.

The Cons and Other segments are divided according to their generation and there may also some permanent segments (as a result of a call to `make-current-allocation-permanent`, `make-object-permanent` or `make-permanent-simple-vector`).

In addition, LispWorks also holds some reserved segments that are used during GC, and `room` prints the size of these too.

The output of `(room t)` also includes the segments for each type. For each segment, it prints the start and end addresses (in hex), the allocated area, and whether there is a free "hole" in the middle of it. For the Large and Static segments, it also prints the generation number of each segment. Permanent Static and Large segments have generation number 3.

See “Mobile GC technical details” on page 154 for more technical details.

**gc-generation**

When the `gen-num` argument is a number, it must be 0, 1 or 2. The value `t` (and `:blocking-gen-num`) is interpreted as 2.

Generation 0 is always promoted, but the `:promote` keyword affects generation 1 and, if non-nil, promotes even if promotion was blocked by `set-promote-generation-1`.

The keyword `:coalesce` is interpreted as in the ordinary 64-bit GC. The keyword `:block` is ignored.

**marking-gc**

Calls `gc-generation` with the `gen-num` argument. It is not useful in the Mobile GC.

**clean-down**

Performs the same GC as `(gc-generation t)`.
reduce-memory

The full argument can be also be \texttt{aggressive}, 0 1 or 2.

count-gen-num-allocation

The \texttt{gen-num} argument can be 0, 1, 2 or 3 (3 means permanent).

in-static-area

This does not affect allocation of conses (which are never static in the Mobile GC).

apply-with-allocation-in-gen-num

The \texttt{gen-num} argument must be 0, 1 or 2.

sweep-all-objects

Does not sweep \texttt{cons} objects in the Mobile GC.

sweep-gen-num-objects

Does not sweep \texttt{cons} objects in the Mobile GC.

The following functions do nothing in the Mobile GC, and the values that they return are not meaningful:

\begin{itemize}
  \item \texttt{set-delay-promotion}
  \item \texttt{set-maximum-segment-size}
  \item \texttt{set-default-segment-size}
  \item \texttt{set-gen-num-gc-threshold}
  \item \texttt{set-blocking-gen-num}
  \item \texttt{gen-num-segments-fragmentation-state}
  \item \texttt{set-spare-keeping-policy}
\end{itemize}
11.5.2 Mobile GC technical details

This section describes the Mobile GC in more detail. For most purposes, you do not need to understand the technical details of the Mobile GC, because it is used automatically and it should just work. You may want to know more if you want to fully understand the output of `room` (especially when called with `T`), and if you want to optimize memory usage (and maybe performance) of the application. In general, you should first use `time` or `extended-time` and `room` or `room-values` to understand the behavior of your application before trying to optimize it.

The ordinary 64-bit GC is "sparse", which means it leaves unused addresses between memory that it has allocated, and it also relies on being able to map memory at specific addresses. The result is a very efficient GC. However, on 64-bit iOS the range of addresses that is available (the address space) is very small compared to other 64-bit architectures (as determined experimentally because Apple do not documented it), and also there is no documented interface for mapping at specific addresses. Therefore the ordinary 64-bit GC cannot work on 64-bit iOS, which is the reason for introducing the Mobile GC.

The Mobile GC is less efficient than the ordinary 64-bit GC, but the only interface that it requires from the underlying OS for memory handling is `malloc` and `free`.

An additional issue specific to iOS is that iOS does not allow execution of machine code that is created dynamically, and the memory region where the code resides is read-only. Therefore the Mobile GC does not support compilation of code in memory at run time. Moreover, functions can contain data that can be modified so this needs to be separated from the code, which is not the case in the ordinary 64-bit memory model. To support this, the images that are used to deliver on iOS are different from the desktop images, though the difference is only in the memory layout of function objects, and from the programmer’s point of view they behave the same. These images differ from the ordinary 64-bit images in that function objects and code are separated, and that function objects are allocated in the same segments as symbols (that is the allocation type `:symbol`). The code is allocated in objects with allocation type `:function`. See “Segments and Allocation Types” on page 147 for more details about allocation types. The names of these images and how to use them are described in “Delivering for iOS” on page 241.
The separation of code and use of the Mobile GC solves two different problems, which in principle could be solved separately. On 64-bit iOS, we have to solve both problems, and therefore the separation of code and the switch to the Mobile GC are done together.

11.5.2.1 Objects alive at delivery time

During delivery for 64-bit iOS, the code is separated out into its own block of memory (the "code block"). Then all of the other objects are put together in a block of memory, which is called the "data block". The data block is divided between non-pointer objects, weak objects and all other objects. The objects in the data block are never GCed, but the GC follows pointers from them to objects allocated at run time. Delivery creates an object file containing the code block and the data block, which is then linked with the rest of the app.

You cannot obtain a pointer to any object in the code block.

`generation-number` returns 3 for objects in the data block.

11.5.2.2 Objects allocated at run time

The Mobile GC has 4 different allocation types (note that these do not match the allocation types of the ordinary 64-bit GC described in “Segments and Allocation Types” on page 147):

- Cons: cons objects
- Static: Static objects
- Large: Very large objects
- Other: All other objects.

The Mobile GC does not allow allocation of static conses. Weak objects are allocated as Other or Large.

The different allocation types are allocated in separate segments, where a segment is a contiguous block of memory. Each allocation type has a variable number of segments, which are printed by `(room t)`. Each non-permanent segment belongs to a specific generation, which can be 0, 1 or 2. The permanent segments, which are created by `make-current-allocation-permanent`,
have generation number 3, even though there is not really a generation 3 (the GC does not collect them).

Like in the ordinary GC, allocation of static objects makes life more difficult for the GC (so it reduces the efficiency of LispWorks), and should be avoided.

Objects that are larger than a threshold (currently 1 MB, but this may change) are allocated in segments with the Large allocation type and are also static.

The vast majority of allocation happens in segments with the Cons and Other allocation types, which are together called "ordinary allocation". The segments for ordinary allocation are all of size 8 MB, including any overhead. For Cons segments, the overhead is larger because conses do not have headers.

The Mobile GC mixes marking and copying techniques. Copying has the advantage of eliminating fragmentation and is also more efficient for typical applications where most allocation is very short lived. On the other hand, it requires spare memory to be available during the GC. Marking creates fragmentation and is slower when most of the objects are freed immediately, but it does not require extra memory. Thus the Mobile GC tries to use copying when possible (that is when it can get enough memory from the operating system), and otherwise uses marking GC. The two methods may be mixed in the same GC operation.

For copying, LispWorks uses reserved segments, which it obtains from the operating system as needed. At the end of the GC, it returns any segments that are no longer needed to the operating system, except for some segments that it keeps in reserve. The amount of reserved memory that it keeps is dynamic, and by default grows as the amount of allocation grows. By default, as long as the amount of memory in ordinary segments is less than 48 MB, LispWorks tries to keep enough reserved segments to copy everything in generation 0 and 1 without asking the operating system for more memory. see set-reserved-memory-policy for details.

The copying GC might promote objects, which means copying them to the next generation. Generation 0 objects that are copied are always promoted (that is copied to generation 1). For Generation 1 objects, it is more complex:
For automatic GC:

`set-promote-generation-1` can be used to block any promotion from generation 1.

If promotion is not blocked (the default), then objects that have already survived a GC of generation 1 are promoted (copied to generation 2) and objects that are new to generation 1 remain in generation 1 (default setting) or are promoted depending on the setting by `set-split-promotion`.

For explicitly invoked GC by a call to `gc-generation`

The keywords `:promote` and `:coalesce` control whether objects from generation 1 are promoted or not.

Generation 2 objects are always copied into generation 2.

Blocking promotion from generation 1 can be used to prevent GCs of generation 2, as discussed in “Preventing/reducing GC of generation 2” on page 159.

### 11.5.2.3 Special considerations for the Mobile GC

Because memory is more limited on mobile platforms, the Mobile GC is tuned to collect its highest generation (2) more often compared to the corresponding operation in the ordinary GC (which is a GC of generation 3). Such a GC may take enough time (in the order of a second) and be frequent enough to annoy users. If that happens then you need to try to tune your application, as described in “Preventing/reducing GC of generation 2” on page 159, and you can also try to reduce the amount that your application allocates.

Very large objects (> 1 MB) that do not contain pointers are handled especially efficiently by the Mobile GC. For example, if your program handles a million small strings of 10-15 characters, then you can save memory and maybe even speed up your program by storing them all in a very large string, and use fixnums to specify the bounds of the small strings within the large string instead of using pointers to the small strings. This saves memory and makes the reduces the work that the GC needs to even if only half of the large string is actually used. Note also that when you finish with it, you can free a very large
object and return its memory to the operating system without doing a GC by calling `release-object-and-nullify`.

When a very large object that may contain pointers (for example a large `simple-vector`) is examined by the GC, it needs to go through all of those pointers. This is wasted work unless either it is long-lived and is rarely seen by the GC, or it is almost full of useful pointers, or if you make it permanent. Objects in general can be made permanent by `make-current-allocation-permanent`, which is discussed in “Preventing/reducing GC of generation 2” on page 159, but very large objects, which are allocated in their own segment, can also be made permanent individually by `make-object-permanent` or `make-permanent-simple-vector`. If most of the elements in a `simple-vector` are not pointers to objects that can be GCed, this substantially improves the performance of LispWorks.

Large objects which are allocated in their own segments can be explicitly freed (releasing the memory they use) by calling `release-object-and-nullify`. That releases the memory without a GC (so it is fast), and works on such objects even if they are permanent.

### 11.5.3 Tuning memory management in the Mobile GC

#### 11.5.3.1 Response to low memory

Mobile platforms typically inform applications when memory availability becomes low. On Android this is done by the `onTrimMemory` or `onLowMemory` methods and on iOS by the `didReceiveMemoryWarning` method. It is probably a good idea to respond to these methods, but it is not essential.

In your implementation of these methods, you should release any system resources that can be released without loss and also try to reduce the memory used by Lisp data. Since the GC sometimes temporarily requires more memory during an operation, it may be a bad idea to do a GC once you get the warning. The function `reduce-memory` is provided to reduce memory usage without requiring more memory temporarily. Note that `gc-generation` can do a much better job than `reduce-memory` in general, but it may require more memory temporarily.
Calling `reduce-memory` with argument `nil` (the default) just releases any reserved memory that LispWorks has kept. It is fast and probably always a good idea. However, with argument `nil`, `reduce-memory` does not perform any GC operation, which in principle could release more memory. Because a GC takes time, it is not obvious whether it worth the trouble.

Calling `reduce-memory` with 0 or 1 causes a GC of generation 0 or 1, which is probably fast enough (unless promotion of generation 1 is blocked and generation 1 grows), but will not typically release much memory.

Calling `reduce-memory` with 2 (or, equivalently, `T`), or even `:aggressive`, can release much more memory, but takes more time, depending on the size of generation 2. Unless it is likely to release a large amount, it is probably not worth it. Thus, unless you know that generation 2 contains a lot of dead objects, you should only call `reduce-memory` with `nil`, or maybe 0 or 1.

If you call `reduce-memory` with a non-nil argument, you should first clear any caches that you have kept, so their contents can be GCed.

To be able to reduce memory usage, `reduce-memory` needs reserved memory to perform a copying GC. Since `reduce-memory` never obtains more memory from the operating system, its effectiveness depends on the amount of reserved memory that it has when it is called. Moreover, any call to `reduce-memory` frees all of the reserved memory (once the GC has occurred if the argument is non-nil), so calling `reduce-memory` with non-nil shortly after a previous call with `NIL` is not going to be effective.

To see how much effect `reduce-memory` had on the memory, you can look at the output of `room` (last line with any argument you give it), or the result of `room-values`. To see how much time it takes, use the `time` macro or `get-internal-real-time`.

### 11.5.3.2 Preventing/reducing GC of generation 2

GC of generations 0 and 1 should normally be fast enough that you do not need to worry about them. GC of generation 2, however, typically takes enough time to be noticeable, and if generation 2 is large (> 100 MB) can take more than a second. Thus you normally want to avoid GC of generation 2.

In a “nicely behaved” application, which we believe is true for most applications, generation 2 never needs to be collected. This is based on the assump-
tion that a nicely behaved application starts with some initialization that allocates long-lived objects, but then enters a "work" phase, where it allocates only short lived objects, which die before they reach generation 2.

Even if there is some "generation leak", that is objects being promoted from generation 1 to 2 that die not long afterwards, the leak may be slow enough that it is not a problem. For example, if your application "leaks" on average 1 kB each second, it would take close to 3 hours of operation to leak 10 MB, which is still too small to worry about (the default minimum size of generation 2 before a GC is 64 MB). So you can usually ignore this kind of leakage and hope that any occasional delay of a second or two after running the application for many hours is not too annoying for the user (though if it only a "generation leak", you can do better by blocking promotion). If you have a leakage of 100 kB per second, the delay would happen every few minutes, which may be too annoying.

To find if your applications leaks to generation 2, you should periodically log the size of either the whole application or of generation 2. The output of room is the most useful thing to log, but you can also use room-values or count-gen-num-allocation. If the application does leak to generation 2, you should determine if it is a real memory leak, which means that the application accumulates live objects, or just a generation leak, which means that objects live long enough to reach generation 2 and then die. To determine that, call (gc-generation T) (or, equivalently, (clean-down)), continue using the application for a while and then call it again. If the leak is just a "generation leak", then the size of generation 2 after (gc-generation t) should stay (more or less) the same. If it grows, then you have a real memory leak.

If your application is "nicely behaved", generation 2, and hence the whole application, will initially grow, typically by few 10’s of megabytes, and then will stay more or less fixed. The size of the whole application will always fluctuate, because generation 0 and 1 fluctuate, but generation 2 should be stable or grow slowly. If this is the case, you probably do not need to do anything further to control memory usage.

If generation 2 does grow, LispWorks will occasionally do a GC of generation 2, which takes a noticeable time (maybe a few seconds if generation 2 is few 100’s of megabytes). If the leak is a real memory leak, it will also cause the application to grow indefinitely.
If the leak is a real memory leak, then the GC cannot do anything about it. One possibility is to make the application run for a limited time, for example by monitoring the size and quitting when it reaches some threshold. If quitting and restarting is possible without much loss, that may be a good solution. Most of applications probably want to avoid that though, in which case you will need to figure out what keeps objects alive and fix it. The functions `sweep-all-objects`, `sweep-gen-num-objects` and `mobile-gc-sweep-objects` can be used to check what kind of objects have accumulated. However, whatever keeps the objects is something in your application, and you will have to find it.

If the leak is only a "generation leak", then there are several ways to deal with it:

- Block promotion from generation 1 to 2 by calling `set-promote-generation-1` with `nil`.
  
  Once you have made this call, the automatic GC will never promote to generation 2 (explicit invocation of the GC ignores this setting). This is useful in situations where the "leaking" objects are live long enough to be promoted to generation 2, or the memory they use is small, so generation 1 does not grow too much. If there are many objects that live longer, then generation 1 will grow, and hence the GC of generation 1 will become slow. You should check if generation 1 grows, but it is probably OK if it remains at 20-30 megabytes allocated after a GC. You can try timing a GC of generation 1 by `(time (gc-generation 1))`. Note that you can switch promotion on and off as needed, so if you can identify phases in your application when allocation is not long-lived and phases when some is long-lived, then you can switch promotion on and off as appropriate.

- Prevent GC in generation 2 by calling `set-generation-2-gc-options`.
  
  Once you have called `set-generation-2-gc-options` with `/minimal-size-for-gc t`, LispWorks will not automatically GC generation 2. It is then your responsibility to GC generation 2 at the appropriate time by calling `(gc-generation 2)`. As above, one of the options is to never GC generation 2, and just quit when the application reaches some size. Otherwise, you will need to identify appropriate points in time to perform the GC.
In an interactive application, you can have a "cleanup" option somewhere that invokes the GC, so the user can invoke it. You probably also want some indicator when the application has grown and needs a "cleanup".

For an interactive application, it may be useful to do a GC when the application becomes backgrounded, but it is not obviously so. The method that is called by the operating system to indicate that the application has been backgrounded must return in a short time, so you probably need to invoke the GC from another thread. Also the operating system may not give much CPU to the application while it is in the background, and may even terminate background applications that take CPU. For example, the "App Programming Guide for iOS" says: "Apps that spend too much time executing in the background can be throttled back by the system or terminated." A GC of a 100-200 megabyte application should not take enough time to cause termination, but it depends both on the underlying system (hardware and OS) and the current state of it, so it is not that predictable. You certainly need to store anything that needs to be stored before doing a GC while in the background.

As long as memory is not constrained, the time it takes to GC generation 2 correlates to the amount alive after the GC rather than before the GC (because it uses copying, so does not touch dead objects). Therefore, if you have points in time in the execution when you know your application uses less memory then these are good points for doing a GC. That would be the case if your application builds a large data structure for a task (allocation of > 10 megabytes), and all this data becomes free when the task finishes. In this situation, it may be useful to perform a GC in the end of the task.

- Tuning the GC of generation 2 by `set-generation-2-gc-options`.

By calling `set-generation-2-gc-options` you can tune the frequency of GC of generation 2. You can either aim for infrequent GCs, which may be long but hopefully rare enough not to be too annoying, or aim for frequent GCs which are fast enough that they do not bother the user.

When the amount that is alive after a GC is almost always much less than the amount alive before, which is quite common, you can tell that to the GC by `set-expected-allocation-in-generation-2-after-`
11.6 Common Memory Management Features

This section summarizes Memory Management functionality common to all LispWorks 7.1 implementations.

11.6.1 Timing the garbage collector

The macro extended-time is useful when timing the Garbage Collector (GC). Use start-gc-timing, stop-gc-timing and get-gc-timing to time GC operations.

11.6.2 Reducing image size

To reduce the size of the whole image, use clean-down.

In 32-bit LispWorks, you can use (clean-down) or the less aggressive (clean-down nil) to reduce the image size when the image is much larger than the amount that is allocated. In 64-bit LispWorks there is no need to do that.

(clean-down t) promotes to generation 3 and tries to reduce the image size, while (clean-down nil) promotes only to generation 2 and does not reduce

gc. This can significantly improve how well the GC copes when it fails to get as much memory as it asks for from the operating system. See set-expected-allocation-in-generation-2-after-gc for details.

- Making the long-lived objects permanent by using make-current-allocation-permanent.

The function make-current-allocation-permanent causes all the currently allocated objects to be made permanent, which means that the GC will not scan or free them in future (but it will still follow pointers from them). That is useful in the typical situation where the application starts with some initialization that creates long-lived objects. Using make-current-allocation-permanent at the end of the initialization makes all these objects permanent, and therefore reduces the time for GC of generation 2. If new objects in generation 2 after initialization are only the result of ”generation leak” then the effect on time can be quite large.
the image size. Experience suggests that the latter is actually more useful in most circumstances.

In some circumstances it is important to avoid enlarging the size of the image even temporarily. The common situation is when the operating system signals low memory. In this situation you should use `reduce-memory` instead of `clean-down`.

### 11.6.3 Allocation of interned symbols and packages

Interned symbols (and their symbol names), and packages, are treated in a special way, because they are assumed to have a long life. They are allocated in the generation specified by the variable `*symbol-alloc-gen-num*`, which has the initial value 2 in 32-bit LispWorks and 3 in 64-bit LispWorks.

Symbols created with `make-symbol` or `gensym` start out in generation 0.

Symbols will be garbage collected if they are no longer accessible (regardless of property lists) but note that in 32-bit LispWorks, if the symbols are in generation 2 then you might need to invoke `gc-generation` explicitly to collect them in a timely manner.

### 11.6.4 Allocation of stacks

Stacks are allocated directly in generation 2 because they are relatively expensive to promote. Therefore creating many processes will cause generation 2 to grow, even if these processes are short-lived.

The variable `*default-stack-group-list-length*` controls the number of stacks that are cached for reuse. Increase its value if your application repeatedly makes and discards more than 10 processes.

### 11.6.5 Mapping across all objects

To call a function on all objects in the image, use `sweep-all-objects`.

### 11.6.6 Special actions

You may want to perform special actions when certain types of object are garbage collected, using the functions `add-special-free-action`, `flag-spe-
cial-free-action, flag-not-special-free-action and remove-special-free-action.

For example, when an open file stream is garbage collected, the file descriptor must be closed. This operation is performed as a special action.

11.6.7 Garbage collection of foreign objects

Users of the Foreign Language Interface may want to specify the allocation of static arrays. The recommended way to do this is to call make-array with :allocation :static. See for example :lisp-array in the LispWorks Foreign Language Interface User Guide and Reference Manual.

11.6.8 Freeing of objects by the GC

Weak arrays and weak hash tables can be used to allow the GC to free objects.


For a description of weak vectors see set-array-weak, page 1012.

11.6.9 Assisting the garbage collector

This section describes techniques that may improve the performance of your application by reducing the GC’s workload.

11.6.9.1 Breaking pointers from older objects

This is a technique that can be useful when older objects regularly point to newer objects in a lower generation. In such a case, when the lower generation (only) is collected these newer objects will be promoted even if the older objects are not live. All of these objects will not get collected until the higher generation is collected.

This is a general issue with generational garbage collection and, if it causes poor performance in your application, can be addressed along these lines. It is not necessarily a problem in every case where older objects point to newer objects.
For example, suppose you are popping items from a queue represented as a list of conses (or other structures), then you can set the “next” slot of each popped item to `nil`.

In the code below, if the `queue-head` cons is promoted to generation $n$, then all the other conses will also be promoted to generation $n$ eventually, until generation $n$ is collected. This happens even after calls to `pop-queue` have removed these conses from the queue.

```lisp
(defstruct queue head tail)
(defun push-queue (item queue)
  (let ((new (cons item nil)))
    (if (queue-head queue)
      (setf (cdr (queue-tail queue)) new)
      (setf (queue-head queue) new)))

(defun pop-queue (queue)
  (pop (queue-head queue)))
```

The fix is to make `pop-queue` set the “next” slot (in this case the `cdr`) of the discarded `queue-head` cons to `nil`, so that it no longer points from an older object to a newer object. For example:

```lisp
(defun pop-queue (queue)
  (when-let (head (queue-head queue))
    (setf (queue-head queue) (shiftf (cdr head) nil))
    (car head)))
```
The LispWorks profiler provides a way of empirically monitoring execution characteristics of Lisp programs. The data obtained can help to improve the efficiency of a Lisp program by highlighting those procedures which are commonly used or particularly slow, and which would therefore benefit from optimization effort.

12.1 What the profiler does

With the profiler running, the Lisp process is interrupted regularly at a specified time interval until the profiler is turned off. Having halted the execution of the process the profiler scans the execution stack and records information about it, including the names of all functions found. A special note is made of which function is at the top of the stack. After profiling stops the profiler can present a report containing a call tree and/or a cumulative columnar report.

The columnar report shows aggregated information about each function as follows:

- The number of times the function was called.
- The number of times the function was found on the stack by the profiler, both in absolute terms and as a percentage of the total number of scans of the stack.
• The number of times the function was found on the top of the stack, both in absolute terms and as a percentage of the total number of scans of the stack.

The call tree shows the name of a root function and a "tree" of callee functions below it. To the right of each function’s name the number of times it was seen on the stack under a particular caller is shown, along with the percentage this represents of the total number of times the function was seen.

The call tree is more computationally expensive to record than the cumulative data. You can choose whether to record and output the call tree, as described in the next section.

12.2 Setting up the profiler

Before a profiling session can start, several parameters must be set, using the function set-up-profiler. If the profiler is invoked before any call to set-up-profiler, it calls set-up-profiler implicitly without any arguments. In many cases that is what you want anyway, and in these cases you do not need to call set-up-profiler, but in some cases you will want to change something.

There are four main areas to consider: the symbols to be profiled, the time interval between samples, the kind of profiling required, and the format of the output.

• By default, all fbound symbols in the image are monitored (and, if KnowledgeWorks is loaded, also all the forward chaining rules). This setting is useful in many cases, but in some cases you will want to see information only about some subset of the symbols, which will make it easier to read the output. Use the keywords :packages and :symbols (and :kw-contexts for KnowledgeWorks) to restrict the set of symbols that will be profiled.

• You might want to specify the time interval between interrupts. The resolution of this value is clearly dependent on the operating system. In most cases the default value, 10ms, is adequate. This number is important, because with these statistical methods of program profiling the accuracy of the results increases with the number of samples taken.
12.3 Running the profiler

On non-Windows systems the kind of profiling required may be set. This refers to what kind of time is monitored in order to determine when to interrupt the Lisp process. There are three possibilities for how the time interval is measured:

- The time the Lisp process is actually executing plus the time that the system is executing on behalf of the process. This is called profile time.
- Just the time that the process is actually executing. This is called virtual time.
- The actual elapsed time, called real time.

The output can be presented as a tree of calls seen and a columnar report (style :tree), or just the columnar report (style :list). You can restrict the data shown in several ways, helping you to focus on the slowest parts of your program.

12.3 Running the profiler

The profiler has two distinct modes. You can use both in the same session, but not at the same time.

To use either mode, you must first call set-up-profiler to load the profiler and set its parameters including the output format.

The macro profile simply profiles all processes while a body of code is run, as described in “Using the macro profile” on page 169. Start profiling this way if you don’t see a need to use the alternate mode.

Alternatively the functions start-profiling, stop-profiling and set-process-profiling offer programmatic control over when profiling occurs and which processes are profiled. This is described in “Programmatic control of profiling” on page 170.

The function do-profiling is a convenience function which allows you to profile multiple threads using start-profiling and stop-profiling.

12.3.1 Using the macro profile

To profile your Lisp forms enter:

(profile <forms>)
This evaluates the forms as an implicit `progn` and prints the results, according to the parameters established by `set-up-profiler`.

Note: you cannot use `profile` (or the graphical Profiler tool) after a call to `start-profiling` and before a call to `stop-profiling` with `print t`, because the two profiling modes are incompatible.

### 12.3.2 Programmatic control of profiling

Your program can control profiling. This is useful when you want to profile only a part of the program.

In your program, call `start-profiling` start collecting profiling information. Call `stop-profiling` with `print nil` to temporarily stop collecting, or call `stop-profiling` with `print t` to stop collecting and print the results. At any point you can call `set-process-profiling` to modify the set of processes for which profiling information is being (or will be) collected.

For example:

```lisp
;; start profiling, current process only
(start-profiling :processes :current)
(do-interesting-work)
;; temporarily suspend profiling
(stop-profiling :print nil)
(do-uninteresting-work)
;; resume profiling
(start-profiling :initialize nil)
(do-more-interesting-work)
;; now, all processes are interesting
(set-process-profiling :set :all)
(do-some-more-interesting-work)
;; stop profiling and print the results
(stop-profiling)
```

Note: you cannot call `start-profiling` inside the scope of the macro `profile` or while the graphical Profiler is profiling, because the two profiling modes are incompatible.

### 12.4 Profiler output

A typical report would be:
profile-stacks called 564 times

Call tree
Symbol seen (%)
1: MOD 17 ( 3)
2: FLOOR 5 ( 1)
1: EQL 8 ( 1)
1: >= 7 ( 1)
2: REALP 2 ( 0)
1: + 6 ( 1)
1: LENGTH 4 ( 1)

Cumulative profile summary
Symbol called profile (%) top (%)
MOD 1000000 17 ( 3) 8 ( 1)
EQL 2000117 8 ( 1) 8 ( 1)
>= 1000001 7 ( 1) 5 ( 1)
+ 1000000 6 ( 1) 6 ( 1)
FLOOR 1000000 5 ( 1) 5 ( 1)
LENGTH 2000086 4 ( 1) 4 ( 1)
REALP 1000001 2 ( 0) 2 ( 0)

Top of stack not monitored 93% of the time

The first line means that Lisp was interrupted 564 times by the profiler.

The call tree shows that in 17 of these interrupts (3% of them) the profiler found the function mod on the stack, in 5 of these interrupts it found the function floor on the stack, and so on. Moreover, floor only appears under the mod branch of the tree, which means that each of these times floor was called by mod.

The cumulative profile summary also shows how many times each symbol was found on the stack. Moreover it shows that the function mod was called 1000000 times, the function eql was called 2000117 times, and so on. (Note: this information is not collected on Intel-based platforms by default.) In 17 of these interrupts it found the function mod on the stack, and on 8 of these occasions mod was on the top of the stack. You can deduce that 526 times the function on the top of the stack was none of those reported.

You can control sort order of the cumulative profile summary with print-profile-list.
12.4.1 Interpretation of profiling results

One important figure is the amount of time it was found on top of the stack in the cumulative profile summary. Just because a function is found on the stack does not mean that it uses up much processing time, but if it is found consistently on the top of the stack then it is likely that this function has a significant execution time. Another thing to check is that you expect the functions near to top of the call tree to take significant time.

It must be remembered that the numbers produced are from random samples and thus it is important to be careful in interpreting their meaning. The rate of sampling is always coarse in comparison to the function call rate and so it is possible for strange effects to occur and significant events to be missed. For example, “resonance” may occur when an event always occurs between regular sampling times, though in practice this does not appear to be a problem.

12.4.2 Displaying parts of the tree

Once profiling information has been recorded, either by stop-profiling or a normal exit from profile, it is possible to print specific parts of the information. The function profiler-tree-from-function prints a tree showing a specific function and the functions called inside it. The function profiler-tree-to-function prints a tree showing a specific function and its callers. This tree is inverted, which means that the children of a node are its callers, rather than callees as in the full tree and the tree printed by profiler-tree-from-function.

12.5 Profiling pitfalls

Profiling should only be attempted on compiled code. If it is done on interpreted code, the interpreter itself is profiled, and this distorts the results for the actual Lisp program.

Macros cannot be profiled as they are expanded during the compilation process. Similarly some Common Lisp functions may be present in the source code but not in the compiled code as they are transformed by the compiler. For example:

```
(member 'x '(x y z) :test #'eq)
```
is transformed to:

\[(\text{memq }'x'((x\ y\ z)))\]

by the compiler and therefore the function \texttt{member} is never called.

Recursive functions need special attention. A recursive function may well be found on the stack in more than one place during one interrupt. The profiler counts each occurrence of the function. Hence the total number of times a function is found on the stack may be much greater than the number of times the stack is examined.

Tail call optimization will prevent the calling function from being found on the stack after the call. You can disable tail call optimization by compiling code with optimize quality \texttt{debug 3}, but note that this might also affect the performance.

Care must be taken when profiling structure accessors. Structure accessors compile down into a call to a closure of which there is one for all structure setters and one for all structure getters. Therefore it is not possible to profile individual structure setters or getters by name.

It must be remembered that even though a function is found on the stack this does not mean that it is active or that it is contributing significantly to the execution time. However the function found on the top of the stack is by definition active, and thus this is the more important value.

It is quite possible that the amount of time the top symbol is monitored is significantly less than 100\% despite the profiler being set to profile all the known functions of the application. This is because at the time of the interrupt an internal system function may well be on the top of the stack.

### 12.6 Profiling and garbage collection

The macro \texttt{extended-time} provides useful information on garbage collection activities.

The \texttt{gc} argument of \texttt{set-up-profiler} controls whether or not the system’s memory management functions are profiled.
12.7 Profiler tree file format

The profiler tree file is produced by calling `save-current-profiler-tree`, or by using the `Save Profiler tree...` item from the `Profiler` menu on the Lisp-Works IDE.

The file contains lines of text encoded UTF-8, to allow it to contain any symbol name.

The first line is handled specially: it must contain the string "LispWorks Profiler Tree" (without the quotes), which confirms that the file is a profiler tree file. In addition, the text following the first colon in the first line, with leading and trailing spaces removed, is the name of the tree.

The remaining lines in the file are the data lines, except those starting with a semicolon which are ignored.

Each data line is divided to 6 fields by a `|` character. The first 5 fields are integers and the last field is an arbitrary sequence of characters (any character except newline). There must be no spaces between the fields.

Each line specifies a node in the tree, in this format:

```
Depth | Count | Call-Count | Seen-Count | Top-Count | Name
```

*Depth* specifies the depth of the node, from which its location in the tree is deduced. The node with depth 0 is the root. For other nodes, the parent of the node is the previous node in the file which has depth smaller by 1. The children of the node, are all following nodes with depth larger by 1, until the next node with a depth less than or equal to *Depth*. In other words, each node is followed by its children in depth-first order.

*Count* is the number of times that the function associated with the node was seen on the call stack within the same branch of the tree, that is with the same chain of callers and on the same process.

*Call-Count, Seen-Count, Top-Count* and *Name* are the function-info of the function associated with the node. Hence they are not specific to the node itself and if the function occurs more than once in the tree (which is common), then copies of the function-info will be present in each occurrence.

The fields of the function-info are:
12.7 Profiler tree file format

**Call-Count**  The number of times that the function was called, if this is recorded. Note that the count is not recorded by default in SMP LispWorks, so it is 0.

**Seen-Count**  The number of times that the function was seen in all the branches of the tree.

**Top-Count**  The number of times that the function was seen at the top of the stack, that is it was actually executing.

**Name**  The name of the function. Note that some nodes do not correspond to actual functions, in which case the name will be a string (including the double quotes).

### 12.7.1 Parsing the file

Because the *function-info* fields of each node are repeated for each occurrence of the same function, it is useful to record *function-info* keyed on the Name. This allows you to associate nodes that are in different branches of the tree but represent the same function.

The name can be read using the Common Lisp reader, provided the currently interned symbols are the same as the interned symbols when the file was produced. Otherwise you may get an error if the package of a symbol does not exist, or if it was external but it is not external when reading. In many cases, just using the name as a string is probably good enough.

### 12.7.2 Viewing the file as text

While it is not possible to see the tree in the file, you can perform simple "queries" just by viewing it in a text editor, for example checking if a function appears anywhere in the tree, finding how often it was visible or getting an idea which function(s) mostly call it.
12 The Profiler
13

Customization of LispWorks

This chapter gives examples of how to make changes to LispWorks to make it more suitable for use by you and your colleagues.

13.1 Introduction

13.1.1 Pre-loading code
You can save an image with changes pre-loaded. This is suitable for changes you want to share with other users of that image, and for code which takes some time to load. It cannot be used to alter settings which the system makes automatically on startup.

“Saving a LispWorks image” on page 179 describes how to do this.

13.1.2 Loading code at start up
You can also load changes each time you start LispWorks. This is suitable for code which loads quickly. For changes only you want to see, put the code in your personal initialization file. For changes to share with other users at your site, put the code in your site initialization file.

“Initialization files” on page 178 describes these initialization files.
13.1.3 Specific customizations

The remainder of this chapter describes some customizations, all of which can be saved in an image or placed in an initialization file, as needed. You can use both techniques: stable code including patches is saved in the image, while experimental or fast-loading code is loaded via the initialization file.

13.2 Configuration and initialization files

There are a number of files that contain configuration and initialization information:

13.2.1 Configuration files

- The LispWorks file `config/configure.lisp` contains many default configuration settings. You can create a customized copy of this file when you install LispWorks, as described in the *LispWorks Release Notes and Installation Guide*.
- The LispWorks file `config/key-binds.lisp` gives the default editor key bindings for Emacs emulation.
- The LispWorks file `config/mac-key-binds.lisp` gives the editor key bindings for Mac OS editor emulation, if supported on your platform.
- The LispWorks file `config/msw-key-binds.lisp` gives the editor key bindings for Microsoft Windows editor emulation, if supported on your platform.

13.2.2 Initialization files

- The LispWorks file `config/siteinit.lisp` is the default site initialization file. The distributed file loads any supplied patches.
- You may also have a personal initialization file which is loaded on start-up. By default LispWorks looks for a file called `.lispworks` in your home directory, although you can change its name and location (see “Setting Preferences” in the *LispWorks IDE User Guide*).
The default location of your home directory varies on Unix systems, but it is typically something like `/home`. On Windows, the directory is constructed from the environment variables HOMEDRIVE and HOME-PATH. The directory itself has the same name as your user name, so if you log on as `john`, your home directory might be `/home/john` on Unix systems or something like `C:\Users\john` on Windows 8.

A sample personal initialization file, the LispWorks file `config/a-dot-lispworks.lisp`, is supplied. You should create a customized copy of this file when you install LispWorks, as described in the *LispWorks Release Notes and Installation Guide*.

### 13.3 Saving a LispWorks image

There are two ways to save an image with changes pre-loaded.

- This section describes the traditional method, using a configuration file and `save-image` script.
- “Saved sessions” on page 182 describes how to save a session, which allows restoring your windowing environment as well as your Lisp objects.

#### 13.3.1 The configuration file

First create a file `my-configuration.lisp` containing the settings you want in your saved image. You may want to change some of the pre-configured settings shown in `config/configure.lisp`, add customizations from the rest of this chapter, or load your application code.

#### 13.3.2 The save-image script

Now create a `save-image` script which is a file `save-image.lisp` containing something like:
(in-package "CL-USER")
(load-all-patches)
(load #+mawindows "~/tmp/my-configuration.lisp"
  #+mawindows "C:/temp/my-configuration.lisp")
#+:cocoa
(compile-file-if-needed
  (sys:example-file
   "configuration/macos-application-bundle")
  :load t)
(save-image
  #+:cocoa
  (write-macos-application-bundle
   "~/Applications/LispWorks 7.1 (32-bit)/My LispWorks.app")
  #--:cocoa "my-lispworks")

The script shown loads my-configuration.lisp from a temporary directory. You may need to modify this.

13.3.3 Save your new image

The simplest way to save your new image is to use the Application Builder tool in the LispWorks IDE. Start the Application Builder as described in the LispWorks IDE User Guide, enter the path of your save-image script in the Build script: pane, and press the Build the application using the script button.

Alternatively you can run LispWorks in a command interpreter and pass your save-image script in the command line as shown below.

- On Macintosh, run in Terminal.app:

  mymac$ "~/Applications/LispWorks 7.1 (32-bit)/LispWorks (32-bit).app/Contents/MacOS/lispworks-7-1-0-x86-darwin" -build save-image.lisp

  Your new application bundle is saved in /Applications/LispWorks 7.1 (32-bit)/My LispWorks.app

- On Microsoft Windows, run in a MS-DOS window:

  C:\temp\"C:\Program Files\LispWorks\lispworks-7-1-0-x86-win32.exe" -build save-image.lisp

  Your new LispWorks image is saved in C:\temp\my-lispworks.exe.

- On Linux, run in a shell:
13.3 Saving a LispWorks image

linux:/tmp$ lispworks-7-1-0-x86-linux -build save-image.lisp

Your new LispWorks image is saved in /tmp/my-lispworks.

For other platforms and for 64-bit LispWorks the image name varies from that shown, but the principle is the same.

13.3.4 Use your new image

Your new LispWorks image contains the settings you specified in my-configuration.lisp pre-loaded.

You can add further customizations on start up via the initialization files mentioned in “Initialization files” on page 178.

Note that your newly saved image runs itself, not a saved session.

13.3.5 Saving a non-GUI image with multiprocessing enabled

To create an image which does not start the LispWorks IDE automatically, make a save-image script, for example in /tmp/resave.lisp, containing:

```
(in-package "CL-USER")
(load-all-patches)
(save-image "~/lw-console"
  :console t
  :multiprocessing t
  :environment nil)
```

Run LispWorks like this to create the new image ~/lw-console:

lispworks-7-1-0-x86-linux -build /tmp/resave.lisp

13.3.6 Code signing in saved images

This section briefly describe when and how LispWorks images are code signed.

13.3.6.1 Signing your development image

On Microsoft Windows and Mac OS X you can sign a development image saved using save-image with the :split argument.
13.3.6.2 Signing in the distributed LispWorks executable

The LispWorks Professional and Enterprise Edition images distributed are not signed, because of the complications around image saving and delivery that this would cause.

The LispWorks for Macintosh Personal Edition application bundle and the LispWorks for Windows Personal Edition executable are both signed in the name of LispWorks Ltd.

13.3.6.3 Signing your runtime application

On Microsoft Windows and Mac OS X you can sign a runtime executable or dynamic library which was saved using \texttt{deliver} with the \texttt{:split} argument.

13.4 Saved sessions

You can save a LispWorks session, which can be restarted at a later date. This allows you to resume work after restarting your computer.

Saving sessions is intended for users of the LispWorks IDE. The graphical tools described in \textit{LispWorks IDE User Guide} provide the best way to use and configure session saving. However it is also possible to save a session programmatically, which is described in this section.

When you save a session, LispWorks performs the following three steps:

1. Closing all windows and stopping multiprocessing.
2. Saving an image. On Mac OS X this creates an application bundle.
3. Restarting the LispWorks IDE and all of its windows.

If a saved session is run later, then it will redo the last step above, but see “What is saved and what is not saved” on page 183 for restrictions.

Sessions are stored on disk as LispWorks images, by default within your personal application support folder (the exact directory varies between operating systems).
13.4 Saved sessions

13.4.1 The default session
There is always a default session, which is used when you run the supplied LispWorks image.

When you run any other image directly, including a saved session or an image you created with \texttt{save-image}, it runs itself (not the default session).

Saved sessions are platform and version specific. In particular, a 32-bit LispWorks saved session cannot be the default session for 64-bit LispWorks, or vice-versa.

13.4.2 What is saved and what is not saved
All Lisp code and data that was loaded into the image or was created in it is saved. This includes all editor buffers, the Listener history and the value of \(*\), \(**\) and \(****\).

All threads are killed before saving, so any data that is accessible only through a \texttt{mp:process}, or by a dynamically bound variable, is not accessible.

All windows are closed, so any data that is accessible only within the windowing system is not accessible after saving a session.

The windows are automatically re-opened after saving the session and all Lisp data within the CAPI panes is retained.

External connections (including open files, sockets, database connections and COM interfaces) become invalid when the saved session is restarted. In the image from which the session was saved, the connections are not explicitly affected but if these connections are thread-specific, they will be affected because the thread is killed. In recreated Shell tools the command history is recovered but the side effects of those commands are not. Debugger and Stepper windows are not re-opened because they contain the state of threads that have been killed.

13.4.3 Saving a session programmatically
You can save a session by calling \texttt{save-current-session}
13.4.3.1 Save Session actions

The first thing that `save-current-session` does is to execute the action-list "Save Session Before".

After redisplaying all the interfaces, the action-list "Save Session After" is executed. That happens both in the saving invocation and the restarting saved image.

13.4.3.2 Non-IDE interfaces

If there are non-IDE interfaces on the screen when `save-current-session` is invoked, there interfaces are destroyed in the first step, and displayed again in the third step. Note that the display will occur in a different thread than the one running the interface before the saving (which was killed in the first step).

If the interface (or any of its children) contains information that is normally destroyed (in some sense) in the `destroy-callback`, this information can be preserved over a call to `save-current-session` by defining methods on the generic functions `capi:interface-preserving-state-p` or `capi:interface-preserve-state`.

13.4.4 Saving a session using the IDE

You can save a session or set up periodic automatic session saving using the configuration tools in the LispWorks IDE. See "Session Saving" in the *LispWorks IDE User Guide* for details.

13.5 Load and open your files on startup

Suppose you always compile and load several files after LispWorks starts. You can arrange for this to happen automatically by adding forms like these in your initialization file:

```lisp
(defvar *my-files*
  '([]/path/to/foo1
     /*path/to/foo2
     /*path/to/foo3"))

(dolist (file *my-files*)
  (compile-file file :load t))
```
If you also want to open these files in the Editor tool, then you can add this form in your initialization file, after those above:

```
(define-action "Initialize LispWorks Tools"
  "Open My Files"
  #'(lambda (screen)
      (declare (ignore screen))
      (dolist (file *my-files*)
        (ed file))))
```

13.6 Customizing the editor

This section explains some of the customizations you can make to the Editor tool in the LispWorks IDE.

13.6.1 Controlling appearance of found definitions

The commands Find Source, Find Source for Dspec and Find Tag retrieve the file containing a definition and place it in a buffer with the relevant definition visible. By default, the start of the definition is in the middle of the Editor window and is highlighted.

The variable `editor:*source-found-action*` controls the position and highlighting of the found definition. The value should be a list of length 2. The first element controls the positioning of the definition, as follows:

- `t` Show it at the top of the editor window.
- A non-negative fixnum Position it that many lines from the top.
- `nil` Position it at the center of the window.

The second element can be `:highlight`, meaning highlight the definition, or `nil`, meaning don't.

For example, to configure the editor so that found definitions are positioned at the top of the window and are not highlighted, do

```
(setq editor:*source-found-action* '(t nil))
```

This variable is set in the file `a-dot-lispworks.lisp`. 
13.6.2 Specifying the number of editor windows

You can specify the maximum number of editor windows that are present at any one time. For example, to set the maximum to 1:

\[
\texttt{(setq editor:*maximum-ordinary-windows* 1)}
\]

This variable is set in the file \texttt{a-dot-lispworks.lisp}.

13.6.3 Binding commands to keystrokes

You can bind existing editor commands to different keystrokes, using \texttt{editor:bind-key}.

The LispWorks file \texttt{config/key-binds.lisp} is supplied. It shows the standard Emacs key bindings for LispWorks.

The following example shows how to rebind ? so that it behaves as an ordinary character in the echo area of tools in the LispWorks IDE — this can be useful if your symbol names include question marks.

\[
\texttt{(editor:bind-key "Self Insert" \\ ? :mode "Echo Area")}
\]

Since ? is then no longer available for help, you may wish to rebind help to Ctrl+?.

\[
\texttt{(editor:bind-key "Help on Parse" Control-? :mode "Echo Area")}
\]

If you use another editor emulation, then see the LispWorks file \texttt{config/msw-key-binds.lisp} or \texttt{config/mac-key-binds.lisp} for the corresponding \texttt{editor:bind-key} forms.

13.7 Finding source code

\textbf{Note:} This section does not apply to LispWorks Personal Edition.

To configure LispWorks so that editor commands such as \texttt{Find Source}, the menu command \texttt{Find Source}, and the dspec system are able to locate definitions in the supplied editor source code:

1. Load the logical host for the editor source code:

\[
\texttt{(load-logical-pathname-translations "EDITOR-SRC")}
\]
2. Configure source finding to know about editor source code:

```lisp
(setf dspec:*active-finders*
    (append dspec:*active-finders*
            (list "EDITOR-SRC:editor-tags-db")))
```

3. Now do (for example) Meta+X Find Command Definition and enter Wfind File.

   The definition of the command Wfind File is displayed in an Editor tool.

See “Controlling appearance of found definitions” on page 185 for information on controlling how the source code is displayed.

## 13.8 Controlling redefinition warnings

By default most system-provided definers such as `cl:defun`, `cl:defmacro`, `cl:defmethod` and so on signal a warning when they redefine an existing definition. You can bind or set `*redefinition-action*` to eliminate such warnings or make it signal error instead.

Also, the system is configured to protect symbols in implementation packages against definition and redefinition. For example, an error is signaled if you attempt to put a function definition on the symbol `cl:*read-base*`. This behavior is configurable by the variables `*handle-warn-on-redefinition*` and `*packages-for-warn-on-redefinition*`. Bear in mind that the default configuration protects the stability of the system, so if you need to prevent such errors it is better to bind one or both of these variables around specific defining forms, rather than setting their global values.

## 13.9 Specifying the initial working directory

The working directory is set on startup and provides the default location for the File > Open... dialog. Call `change-directory` in your initialization file (see “Initialization files” on page 178) to control the initial working directory.
13.10 Using ! for :redo

The default way of redoing the previous command from the command history is via :redo. If you want to use ! (exclamation mark) instead of :redo, add the following to your .lispworks file:

```
(set-macro-character #\!
    #'(lambda (stream char)    
        ':redo))
```

You may wish during some sessions to reset ! back to its normal role as a character. To do this, evaluate:

```
(set-syntax-from-char #\! #\@)
```

13.11 Customizing LispWorks for use with your own code

This section contains some information on customizations you can make in order to make developing your own code a little easier.

13.11.1 Preloading selected modules

If you frequently use some code that is normally supplied as separate modules, you can load them at start-up time from your initialization file. This file is called .lispworks by default, but can be changed to be any other filename. See “Setting Preferences” in the LispWorks IDE User Guide for details.

For example, to load the dynamic-completion code every time you start LispWorks, include the following in your initialization file.

```
(require *dynamic-complete*)
```

13.11.2 Creating packages

When writing your own code that uses, for instance, the capi package, create a package of your own that uses capi — do not work directly in the capi package. By doing this you can avoid unexpected name clashes.
13.12 Structure printing

By default defstruct generates a method on print-object. You can avoid this by binding at macroexpansion time the variable structure:*defstruct-generates-print-object-method*.

13.13 Configuring the printer

This section applies only on non-Windows platforms when running the Motif IDE.

You can configure your LispWorks image for your printer, by selecting File > Printer Setup from any tool with printing capacities, for example the editor, and choosing Add Printer.

When configuring a printer, the CAPI printing library prompts for a PostScript Printer Description file (PPD), which defines such things as the paper size and the printable area of the page, in the form of a standard PostScript language header. The printing code splices this file into the PostScript produced from submitting a CAPI printing request.

The library on the LispWorks CD contains a generic PPD file, called generic.ppd, that defines these values conservatively to ensure that it should work with most printers. For accurate results, you should use the PPD supplied with your printer.

The PPD files are placed in the ppd subdirectory of the postscript directory in the lispworks library directory. Files added to the ppd directory are expected to have the extension " .ppd".

13.13.1 PPD file details

A PPD file contains a description of the attributes and capabilities of a given printer, such as paper sizes supported, the printable area of the page, the number and names of input paper trays, optional features such as additional paper trays or duplex units, and so on, together with the printer-specific PostScript language commands necessary to use the features.

The file generic.ppd defines a simple generic printer supporting A4, A3, US letter, and US legal paper sizes, and supporting manual feed. It defines con-
servative margins (1 inch all round), and the documents generated should be compatible with most PostScript printers. It is suitable for producing PostScript files when the destination printer is unknown, and may also be used if the appropriate PPD for the printer is not available.

However, for the best results, we recommend the use of the appropriate PPD for the printer. This allows you to specify which optional features (if any) have been installed on the printer, and ensures that the Print dialog provides access to appropriate printer capabilities such as multiple input trays and duplex printing. This also ensures that the CAPI uses the correct values for the printable areas of the page.
This chapter describes how to create a dynamic library or DLL from LispWorks and discusses use of the library.

14.1 Introduction

You can use LispWorks to build a dynamic library on Microsoft Windows, Intel Macintosh, Linux, x86/x64 Solaris and FreeBSD.

To do this, use save-image or deliver and supply a list value for dll-exports. On platforms other than Windows passing dll-added-files also creates a dynamic library.

The result is a library that cannot be executed on its own, but can be dynamically loaded by another process. On Windows this is done with the Windows APIs LoadLibrary and then GetProcAddress. On other platforms the dynamic library can be loaded by dlopen and then dlsym.

The dynamic library is usually of file type dll on Windows, dylib on Macintosh and so on Linux, x86/x64 Solaris or FreeBSD. The first implementation of this functionality in LispWorks was on Microsoft Windows only, therefore the terminology that is used is sometimes Windows-like. In particular “DLL” refers to any dynamic library.
A program that loads a LispWorks dynamic library must be compiled and linked as follows:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Link Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>Link with libpthread.so</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>Link with libpthread.so</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>No special requirements</td>
</tr>
<tr>
<td>Solaris</td>
<td>Compile and link multithreaded (for example, using the -mt option to Oracle’s cc)</td>
</tr>
</tbody>
</table>

### 14.2 Creating a dynamic library

To deliver a LispWorks runtime as a dynamic library supply a list value for `dll-exports` when calling `deliver`.

To save a LispWorks image as a dynamic library supply a list value for `dll-exports` when calling `save-image`.

Additionally on Linux, x86/x64 Solaris, Macintosh and FreeBSD platforms, you can supply a list value for `dll-added-files` to deliver or save a dynamic library.

**Note:** a LispWorks dynamic library is licensed in the same way as a LispWorks executable.

#### 14.2.1 C functions provided by the system

When LispWorks is a dynamic library the functions described in Chapter 54, “Dynamic library C functions” are automatically available. They allow the loading process control over relocation and unloading of the library.

#### 14.2.2 C functions provided by the application

`dll-exports` specifies application-defined exported functions in a LispWorks dynamic library.

Exports can also be provided in the files named in `dll-added-files`, on Linux, x86/x64 Solaris, Macintosh and FreeBSD platforms.
14.2.3 Example

This script saves an image `hello.dll` which is a Windows DLL:

```lisp
(in-package "CL-USER")
(load-all-patches)
;; The signature of this function is suitable for use with
;; rundll32.exe.
(fli:define-foreign-callable ("Hello"
    :calling-convention :stdcall)
  ((hwnd w:hwnd)
   (hinst w:hinstance)
   (string :pointer)
   (cmd-show :int))
  (capi:display-message "Hello world"))

(save-image "hello"
  :dll-exports '("Hello")
  :environment nil)
```

Run the script by

```
lispworks-7-1-0-x86-win32.exe -build hello.lisp
```

on the command line, or use the Application Builder tool.

(See “Saving a LispWorks image” on page 179 for more information about how to save an image.)

You can test the DLL by running

```
rundll32 hello.dll,Hello
```

on the command line.

To see the dialog, you may need to dismiss the LispWorks splashscreen first.

14.3 Initialization of the dynamic library

Each of the exports specified via `dll-exports` ensure first that LispWorks has finished initializing. If initialization has not yet started, they start the initialization process themselves. This is true regardless of the value of `automatic-init` (see below).
A LispWorks dynamic library is initialized automatically on loading, or not, according to the value of *automatic-init* in the call to `deliver` or `save-image`.

### 14.3.1 Automatic initialization

On Microsoft Windows when *automatic-init* was true the initialization finishes before the Windows function `LoadLibrary` returns, and if LispWorks fails for some reason then the call to `LoadLibrary` fails too.

On other platforms when *automatic-init* was true, during the automatic initialization `dlopen` just causes the initialization to start and returns immediately. The initialization will finish sometime later. The LispWorks function `LispWorksState` can be used to check whether it finished initializing.

Automatic initialization is useful when the dynamic library is something like a server that does not communicate by function calls. On Windows it also allows `LoadLibrary` to succeed or fail according to whether the LispWorks dynamic library initialized successfully or not.

### 14.3.2 Initialization via InitLispWorks

Not using automatic initialization (that is, creating the dynamic library with *automatic-init* `nil`) allows using `InitLispWorks` to relocate the image if necessary, and do any other initialization that may be required.

### 14.4 Relocation

LispWorks normally maps its heap on startup in the same place that it was when it was saved, and when it needs more memory it maps this nearby. This applies when LispWorks is a dynamic library as well as for LispWorks executables.

This mapping can cause memory clashes with other software, which may be avoided by relocating LispWorks. Most of the LispWorks implementations are relocatable though the details vary between platforms and between 32-bit LispWorks and 64-bit LispWorks.

On Microsoft Windows and Macintosh, LispWorks detects and avoids memory clashes automatically. On other platforms, you can relocate a LispWorks dynamic library (for all the relocatable implementations) if necessary by a
suitable call to `InitLispWorks` as described in “Startup relocation” on page 459.

### 14.5 Multiprocessing in a dynamic library

Multiprocessing is started automatically in a LispWorks dynamic library. Therefore you can arrange for Lisp initialization operations by adding process specifications to `*initial-processes*`.

For example, if you have a function like this:

```lisp
(defun my-server ()
  (let ((s (establish-a-socket)))
    (loop (accept-connection s))))
```

you need to do something like:

```lisp
(pushnew '(*My server* () my-server) mp:*initial-processes*
 :test 'equalp)
```

before saving or delivering your library.

### 14.6 Unloading a dynamic library

Before a LispWorks dynamic library is unloaded, LispWorks should be made to ‘quit’ cleanly, allowing it to clean up resources that it uses.

When the LispWorks dynamic library is loaded by a main process which you (the LispWorks programmer) do not control, then use `dll-quit`. If you control the main process, then use `QuitLispWorks` instead. For the details, see the respective manual entries for `dll-quit` and `QuitLispWorks`. 
LispWorks as a dynamic library
The LispWorks Java interface allows you to:

- Define "Java Callers" which are Lisp functions that call Java methods or constructors, or access Java fields. You can either define specific callers, or "import a Java class", which means automatically generating callers for all the class public methods, constructors and fields.

- Make and access Java arrays.

- Make calls from Java into Lisp, either by calling Lisp directly or making proxies that implement some Java interface ("Lisp proxy"), and using a Lisp proxy where Java requires an object that implements an interface.

- Access Java objects.

- Integrate Java in a limited way with CLOS.

- Make socket streams using Java sockets. See “Socket streams with Java sockets and SSL on Android” on page 432.

Calling into Java using the callers and accessing arrays does not require any specific Java code. Calling from Java into Lisp requires having the com.lisp-works.LispCalls class (supplied as a JAR file), and using methods from it.

The Java interface is a module which needs to be loaded by calling

```lisp
(require "java-interface")
```
The Java interface symbols are exported from the package LW-JI, documented in Chapter 41, “The LW-JI Package”.

The Java interface requires Java edition 6 or later.

15.1 Types and conversion between Lisp and Java

15.1.1 Mapping of Java primitive types to and from Lisp types

The 8 primitive Java types map naturally to Lisp types:

Table 15.1 Mapping from primitive Java types to Lisp types

<table>
<thead>
<tr>
<th>Java</th>
<th>Lisp</th>
</tr>
</thead>
<tbody>
<tr>
<td>long, int, short, byte</td>
<td>integer</td>
</tr>
<tr>
<td>double</td>
<td>double-float</td>
</tr>
<tr>
<td>float</td>
<td>single-float</td>
</tr>
<tr>
<td>char</td>
<td>integer</td>
</tr>
<tr>
<td>boolean</td>
<td>(member t nil)</td>
</tr>
</tbody>
</table>

The mapping from Lisp to Java is not always obvious, for example because a Lisp integer can map to long, int, short, char or byte. In most cases, like method calls, the target Java type is known. In these cases, LispWorks allows integer in the acceptable range for byte, short, int, long and char, any Lisp float for float and double, t and nil for boolean.

When the target is not known, like storing a value in a Java array object (that is type java.lang.Object[]) or using lisp-to-object, LispWorks uses this mapping:

Table 15.2 Mapping from Lisp when target Java type is unknown

<table>
<thead>
<tr>
<th>Lisp</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers that fit in 32 bits</td>
<td>int</td>
</tr>
</tbody>
</table>
15.1 Types and conversion between Lisp and Java

LispWorks has a set of keywords and FLI types to match the primitive types, which can be used to specify these types, for example as the type of an array. The keyword names are the Java name (uppercased), and the FLI type names are the Java name preceded by J (and uppercased), exported from LW-JI. These are shown in the table below.

Table 15.3 Keywords and FLI types matching primitive types

<table>
<thead>
<tr>
<th>Java type</th>
<th>Keyword</th>
<th>FLI type</th>
<th>Underlying FLI type</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>:short</td>
<td>jshort</td>
<td>:short</td>
</tr>
<tr>
<td>long</td>
<td>:long</td>
<td>jlong</td>
<td>:int64</td>
</tr>
<tr>
<td>byte</td>
<td>:byte</td>
<td>jbyte</td>
<td>:byte</td>
</tr>
<tr>
<td>char</td>
<td>:char</td>
<td>jchar</td>
<td>(:unsigned :short)</td>
</tr>
<tr>
<td>double</td>
<td>:double</td>
<td>jdouble</td>
<td>:double</td>
</tr>
<tr>
<td>float</td>
<td>:float</td>
<td>jfloat</td>
<td>:float</td>
</tr>
<tr>
<td>boolean</td>
<td>:boolean</td>
<td>jboolean</td>
<td>(:unsigned :char)</td>
</tr>
<tr>
<td>int</td>
<td>:int</td>
<td>jint</td>
<td>:int</td>
</tr>
</tbody>
</table>

Note: The Java type char (and hence the class Character) corresponds to UTF-16 code units, which is equivalent to unsigned short. It does not corre-
spond to Unicode characters, and therefore cannot be mapped to LispWorks characters.

### 15.1.2 java.lang.String

LispWorks deals specially with `java.lang.String` objects, converting them automatically to Lisp strings when receiving them (return value of methods or arguments to calls into Lisp), and converting Lisp strings to `java.lang.String` when passing them (argument to method calls, return values from calls into Lisp). It is therefore possible to think of strings as another primitive type. The overhead associated with this conversion for short strings (tens of characters) is smaller than the overhead associated with passing a Java non-primitive object. Even for larger strings, the fact that all the data in the string is passed in one call without further Java/Lisp interaction make it an effective way of passing data.

### 15.1.3 Java non-primitive objects

All Java non-primitive objects are represented in LispWorks as foreign pointers of type `jobject`. `jobject` is a proper Lisp type, that is you can use `cl:typep` and specialize methods on it. The actual Java class of the object is not consistently represented, unless you explicitly ask for it using `jobject-class-name`. You can get a string describing the Java object in the way that Java "thinks" (that is the result of `toString`) using `jobject-string`. If you need a Java null value, then you can use the constant `*java-null*.`

Instances of `standard-java-object` are also considered to represent Java objects. `standard-java-object` instances have a slot that contains the actual `object`, which is used when an instance of `standard-java-object` is passed to the interface functions. In the text below, when argument is specified as "java-object" or "Java object", it can be either a `object` or an instance of `standard-java-object`.

### 15.2 Calling from Lisp to Java

To call into Java you typically define Java callers. There are two possible approach for defining the callers: importing classes or defining specific callers.
In addition, you can call Java methods and constructors and access Java fields without defining them first.

### 15.2.1 Importing classes

Importing a Java class means that the system generates definitions for all the public methods, constructors and fields for this class. For example, to generate and evaluate the definition, execute:

```lisp
(import-java-class-definitions "java.io.File")
```

And to write the definitions to a file:

```lisp
(write-java-class-definitions-to-file "java.io.File" "filename.lisp")
```

The import macros and functions all take various keyword arguments to control exactly what they generate.

*import-java-class-definitions* would normally appear as a top level form in your source file, and when the file is loaded it generates all the definitions. *write-java-class-definitions-to-file* can be used to generate all the definitions and write them to a file, which is an ordinary Lisp source file that can be compiled and loaded as usual. There is also *write-java-class-definitions-to-stream*, which writes the definitions to a stream, and *generate-java-class-definitions*, which returns a list of the definitions, which may be useful sometimes (they are actually used by *import-java-class-definitions* and *write-java-class-definitions-to-file*).

The actual definitions that the importing interface generates are the same as you would write yourself, using the appropriate defining forms: *define-java-caller*, *define-java-constructor*, *define-field-accessor*. These are discussed further in “Defining specific callers” on page 202.

Importing has the obvious advantage that you do not need to type all the method names. It has two disadvantages:

- The generation of the definitions relies on having access to the class definition and running Java virtual machine (JVM), which may or may not be a hassle. For example, if your code contains an *import-java-class-definitions* form, it will need the JVM running and the class defini-
tion accessible when it is loaded as source or when it is compiled (loading the binary file does not require Java).

If the requirement for Java is an issue, you can work around it using `write-java-class-definitions-to-file` (or use `write-java-class-definitions-to-stream`), and use the resulting file as your source code. The call to `write-java-class-definitions-to-file` requires Java, but you need to do it only once, and it can be on a different computer to the one you develop on. For a public class (standard Java, standard Android) you can even ask Lisp Support to create the file for you. This approach also allows you to edit the definitions if you have any reason to. The definitions also contain the signatures of all the methods and constructors.

- The other disadvantage of importing is that it "pollutes" your namespace with many definitions, of which you may be needing only a few. To reduce the chances of clashes, the default setting creates a Lisp package for each Java package, and uses a unique name for the package. This makes the code less Lisp-like. Using the keywords to import interface allows you some control on the naming that it uses.

If you deliver your application without shaking (the default for levels 0 and 1), using import will also cause your application to be larger than it needs to be. If you import many classes this difference may be significant. If you deliver with shaking (default for level 2 or higher), the callers that are not used will get shaken out and so will not affect the size of your application.

### 15.2.2 Defining specific callers

You define specific callers by using the various definers, which are typical defining macros, but the body is automatically generated:

```lisp
define-java-caller
Defines a caller that calls a Java method.

define-java-callers
Defines several method callers for the same class.

define-java-constructor
```
15.2 Calling from Lisp to Java

Defines a caller that calls a Java constructor method.

\textbf{define-field-accessor}

Defines callers to access (read and write) a field.

In addition, you can define callers dynamically at run time using the \texttt{setup-\*} functions \texttt{setup-java-caller}, \texttt{setup-java-constructor} and \texttt{setup-field-accessor}, which are functions that match the \texttt{define-\*} macros above.

The \texttt{setup-\*} functions effectively do exactly what the \texttt{define-\*} macros do, but the code looks nicer with the macros, and the LispWorks Editor can find your definitions.

Methods and constructors are similar enough that they are described here together. Constructors are by definition always "static" in the Java terminology.

Defining a caller for a method or constructor defines a Lisp function that when called invokes the Java method. The Java method is supplied by its class name and name (except constructors, which implicitly map to the constructor methods of the class), which means that there may be more than one Java methods or constructor that are applicable.

For example:

Define a Lisp function \texttt{my-probe-file} which invokes the Java method \texttt{java.io.File.exists}:

\begin{verbatim}
  (define-java-caller my-probe-file "java.io.File" "exists")
\end{verbatim}

Define a Lisp function that calls one of the constructors of \texttt{java.io.File}:

\begin{verbatim}
  (define-java-constructor my-make-file "java.io.File")
\end{verbatim}

At run time, \texttt{my-make-file} will check which of the constructors of \texttt{java.io.File} matches the arguments, and then call it.

See “Actual Java call” on page 205 for a description of how the callers actually work.

Defining a field accessor defines a Lisp function that reads the field value, potentially another Lisp function to set the value (if it is not final), and a symbol macro that expands to calls to the getter or setter. For ordinary (non-static)
fields, the getter needs to be called with the object from which to read the value, and the setter must be called with the object and the value. For static fields, the getter takes no arguments, and the setter takes the new value.

15.2.3 Verifying callers

Compared to importing classes, explicit definitions have the advantages that they do not need Java running until run time, you define only the callers you need, and you select the names of the Lisp functions. The main disadvantage is that you have to type much more, and that you may have typing errors in the method names which are not reported during compilation.

The functions verify-java-caller and verify-java-callers are provided as a way to guard against such typing errors. These functions need Java running, and they check whether the callers have matching Java methods, and return information about missing methods. The intention is that at least during development, you will call verify-java-callers at the beginning of the application and log the result, which will allow you to check whether any method is missing. It may also be useful if you use classes whose definitions may change, for example when the Java code and Lisp code are developed in parallel, or when you use non-standardized Java code.

verify-java-caller and verify-java-callers force the caching of run time information that the callers normally do in their first call.

15.2.4 Calling methods without defining callers

You can call Java methods by passing the full method-name as a string "package.class.method" to call-java-method. The actual run time behavior is as described in “Actual Java call” on page 205.

Note: call-java-method caches the relevant information using the string as the key, while properly defined callers close over it. Therefore call-java-method is slightly slower, but the difference is not significant. The only significant difference is that you can verify the caller to check against typing errors, while with call-java-method you will find a typing error in the method or class name only when you call it. If you find using call-java-method convenient and do not need the verification, there is no reason not to use call-java-method in preference to defining the callers explicitly.
15.2 Calling from Lisp to Java

You can construct an object of a class by calling create-java-object, supplying the full class name followed by the arguments to the constructor. The actual run time behavior is as described in “Actual Java call” on page 205.

You can access fields without defining accessors using read-java-field and set-java-field. There is also checked-read-java-field, which is like read-java-field but does not error on failure, check-java-field to check whether a field exists, and java-field-class-name-for-setting to find the class of the value.

15.2.5 Actual Java call

When a Java caller is called the first time or a call without definition is done and not cached yet, the function finds the relevant method(s), their arguments and return value types, and caches it (see verify-java-caller or verify-java-callers for pre-caching). That includes finding the class, and then finding the relevant methods or constructors. It then uses this information to decide which method is applicable, how to convert the argument to Java where needed, and how to convert the return value back to Lisp. It also decides which JNI function to use to perform the actual call.

Before doing the call LispWorks checks whether the arguments are of the correct type, and in most of the cases can catch and give Lisp errors as appropriate before calling into Java.

For an ordinary (non-static) Java method, the arguments to the Lisp function must start with the actual Java object for which the method needs to be applied. The rest of the arguments to the Lisp function are passed to the method. Thus the number of the arguments to the Lisp function needs to be one more than the number of (explicit) arguments to the Java method. The invocation is virtual (normal Java invocation), which may mean that the actual Java method that is ultimately executed may be defined in a subclass of the class that passed to the definer, if the object belongs to this subclass.

For static Java methods (including constructors) the given argument list is passed to the method.
15.3 Calling from Java to Lisp

Calling from Java to Lisp requires the Java class `com.lispworks.LispCalls`, and Java code that uses methods from this class. Currently all methods are static. `com.lispworks.LispCalls` is supplied by LispWorks in the file `7-1-0-0/etc/lispcalls.jar`, except on Android where it is part of the `7-1-0-0/etc/lispworks.jar` file. After Java is initialized, either by an explicit call to `init-java-interface` or implicitly by the system (for example on Android), you can check whether the Java to Lisp calls are possible (the class `LispCalls` is available) by using `check-lisp-calls-initialized`.

There are two mechanisms for calling from Java to Lisp: direct calls and using proxies. Direct calls means calling directly a Lisp function from Java, passing the name of the symbol to `funcall` and the arguments. Using proxies meaning creating proxies from Lisp, and then passing such Lisp proxies to places where the interface(s) that it implements are required. Invoking a method on such proxy ultimately calls a Lisp function.

Direct calls are simple to use, and if you have a simple Java/Lisp interface can be all that you need. The proxies are needed when you use somebody else’s interface, for example implement callbacks to user interaction in Android. They are also useful even if you write the Java side too to make a cleaner interface on the Java side, which is easier to switch between different implementations.

15.3.1 Direct calls

You can make direct calls from Java to Lisp using one of the `Call<type>[VA]` static methods from `LispCalls`, which have these signatures:
public static int callIntV(String name, Object... args)
public static int callIntA(String name, Object[] args)
public static double callDoubleV(String name, Object... args)
public static double callDoubleA(String name, Object[] args)
public static Object callObjectV(String name, Object... args)
public static Object callObjectA(String name, Object[] args)
public static void callVoidV(String name, Object... args)
public static void callVoidA(String name, Object[] args)

The <type> in Call<type>[VA] specifies the return type, and V or A specify whether the arguments are supplied as Variable arguments or Array. Otherwise the pairs of V and A methods behave the same.

All these methods apply the Lisp symbol which is named by the name argument to the arguments supplied by the Array or the Variable arguments, and return the result.

Note that on the Lisp side you will need to keep the Lisp symbol when delivering, most conveniently by hcl:deliver-keep-symbols (see the LispWorks Delivery User Guide), and the name of the symbol is not interpreted using cl:read.

See com.lispworks.LispCalls for full details.

15.3.2 Using proxies

Using proxies allows you to create from inside Lisp a Java proxy which implements one or more Java interfaces. The proxy can then be used whenever an object that implements any of the interfaces is required. When a method is applied to a proxy, it ultimately calls a Lisp function.

Creating a proxy in Lisp is done in two steps:

1. Defining a proxy, specifying
   i) A name (a symbol).
   ii) The interfaces that it implements.
   iii) The Lisp functions that get called for each method.
iv) A default function.

v) Several other options.

Above, i) and ii) are obligatory, the other steps are optional.

Defining a proxy is done normally at load time by `define-lisp-proxy`. It is possible to define a proxy at run time using `setup-lisp-proxy`. For example, defining a proxy that implements the `onTouchListener` interface, specifying that when the method "onTouch" is invoked it causes the function `text-view-on-touch-callback` to be called:

```lisp
(define-lisp-proxy my-text-view-on-touch-proxy
  ("android.view.View.OnTouchListener"
   ("onTouch" text-view-on-touch-callback)))
```

2. Making a proxy object using the name of a proxy definition by `make-lisp-proxy` or `make-lisp-proxy-with-overrides`, or by calling inside Java the method `com.lispworks.LispCalls.createLispProxy`. The result of making a proxy is a Java proxy object, which can be used in Java. For example, assuming the definition above and that you have a `View` in `mTextView`:

```java
Object listener = LispCalls.createLispProxy("MY-TEXT-VIEW-ON-TOUCH-PROXY");
// Check the type of listener to allow for errors in Lisp
if (listener instanceof View.OnTouchListener)
    mTextView.setOnTouchListener((View.OnTouchListener)listener);
```

This will cause the Lisp function `text-view-on-touch-callback` to be called whenever the `View` in `mTextView` is touched.

**Note**: the result of `make-lisp-proxy` or `make-lisp-proxy-with-overrides` is "local", which means that it cannot be used outside the dynamic scope of the call to Lisp from Java in which it was created. If it is created outside the scope of a call from Java to Lisp, it must be used only in the thread that it was created.

When defining a proxy, you do not need to specify all the methods. You can specify a default function, which is called for any method for which you did not specify a function. See for example the proxy `lisp-othello-server-lazy` in this example, which does not specify any method, and instead specifies a default function that handles all of them:
When defining a proxy, it is also possible to specify that the Lisp functions should be called with an extra argument user-data, which is associated with each specific proxy by passing :user-data to make-lisp-proxy or make-lisp-proxy-with-overrides. This allows you to link each proxy with some of your data. If you do not specify this option, the functions in the proxy need to use the arguments and global data to decide what to do.

It is also possible to "override" the Lisp function at run time, which means specifying that when a Lisp function for a method should be invoked, another function is invoked instead. Overriding is specified by passing either of :overrides or :overrides-plist to make-lisp-proxy, or by using make-lisp-proxy-with-overrides. The main advantage of overriding is that it allows you to use run time closures, while the proxy definition itself allows only symbols. Overrides are efficient and are simple to use. For example, with the definition above, you can override the callback by:

```lisp
(let ((closed-something (creating-something)))
  (make-lisp-proxy 'my-text-view-on-touch-proxy
                  :overrides-plist
                  (list 'text-view-on-touch-callback
                        #'(lambda (&rest args)
                             (apply 'callback-with-something closed-something args))))))
```

which will cause a touch to invoke callback-with-something on closed-something and args.

Note that this example could easily be done using :user-data instead, but that will have to be specified "statically" in the proxy definition, while overriding can all done dynamically when creating individual proxies.

The Java method com.lispworks.LispCalls.createLispProxy cannot do overriding, it must be done inside Lisp by make-lisp-proxy or make-lisp-proxy-with-overrides.

To make it easier to detect typing errors in specifying the interface names and method names or specifying a Lisp function, the functions verify-lisp-proxies and verify-lisp-proxy are provided to verify all proxies or only one, respectively. Verification checks that all the specified functions are actually defined, and optionally also that all the methods that are declared in the
interfaces are defined. The latter check must be done with Java running. You will typically use it when starting the application to check that all the proxies are OK, at least during the development phase.

The Lisp functions of the proxy are ordinary Lisp, but they need to return the correct value, unless the method has `Void` as its return type. Returning the wrong value will call the `java-to-lisp-debugger-hook` (see `init-java-interface`) with an appropriate condition, and then return zero of the correct type (that is 0, 0d0, 0f0, Java `false`, or Java `null`) from the method call.

The call to the Lisp function is wrapped such that trying to throw out of it does not actually finish the throw, and instead returns zero of the correct type from the method call.

In some cases the method needs to throw some exception. The function `throw-an-exception` can be used to throw an exception from inside a call to a proxy function.

### 15.4 Working with Java arrays

Java arrays are represented inside Lisp by a `jobject` or an instance of `standard-java-object`, like any other Java object. The function `java-array-element-type` returns the element type of a Java array or `nil` if it is not an array, and it is fast enough that it can be used as a predicate to determine whether a `jobject` represents an array.

`java-array-length` returns the length of a Java array.

`java-primitive-array-element-type` and `java-object-array-element-type` return the same values as `java-array-element-type` for an array of primitive type or an array of non-primitive type respectively, otherwise they return `nil`. They are fast and can be used as predicates to decide whether an array is of primitive type or not.

Java arrays of higher dimensions are represented recursively as vectors of vectors, which affects the way you use the accessors.

#### 15.4.1 Accessing a single element

The accessor `jvref` can be used to get and set (with `cl:setf`) the value in a Java "Vector" (that is, a one-dimensional array). For a multi-dimensional array,
15.4 Working with Java arrays

`jvref` gets and sets the first level "Vector", in other words it returns another array of one less dimension.

`jaref` can be used to get and set elements of arrays with any number of dimensions. If the number of dimensions given is less than the rank of the array, it gets or sets the corresponding sub-array.

Both `jvref` and `jaref` can be used to access arrays of any type. `jvref` is slightly faster, and does not allow passing wrong number of arguments.

**Note:** when accessing an element of a multi-dimensional array, `jaref` needs to get the sub-arrays for the sub-dimensions. This means it is relatively inefficient when used to access elements in the same sub-array. It is more efficient to get the sub-array and access it. For example, instead of

```lisp
(dotimes (z 10)
  (do-something (jaref java-array 3 4 z)))
```

use

```lisp
(let ((sub-array (jaref java-array 3 4)))
  (dotimes (z 10)
    (do-something (jvref sub-array z))))
```

Assuming `java-array` is not a primitive array, it is even better to use the multiple access functions:

```lisp
(let ((sub-array (jaref java-array 3 4)))
  (map-java-object-array 'do-something sub-array :end 10))
```

15.4.2 Making Java arrays

The function `make-java-array` is used to make Java arrays of any rank and type. It takes as first argument a class specifier, followed by the dimension(s). The class specifier specifies the type of the elements in the array, which may be any type (both primitives and proper classes).

It is also possible to create primitive arrays with data copied from Lisp arrays using `lisp-array-to-primitive-array`. 
15.4.3 Multiple access functions

The multiple access functions are used to access elements in one-dimensional arrays ("Vectors"). They are much more efficient than accessing each element separately.

Multiple access of primitive and non-primitives is done in a different way: non-primitive arrays are accessed by `map-java-object-array`, which maps a function on the objects in the array. Primitive arrays are accessed by `primitive-array-to-lisp-array` and `lisp-array-to-primitive-array` (copy to or from a Lisp array) or `get-primitive-array-region` and `set-primitive-array-region` (copy to or from a foreign array). String arrays are regarded as `Object` arrays for this distinction.

`map-java-object-array` maps a function across an array. It has keyword arguments to control the actual operation, including specifying the range and direction, writing back the result of the call, and collecting the values. When called on multi-dimensional arrays, `map-java-object-array` accesses the top level elements, that is sub-arrays of one less dimension.

`primitive-array-to-lisp-array` and `lisp-array-to-primitive-array` take a Java primitive array or a Lisp array respectively, and copy the elements to a Lisp array or Java primitive array. Both functions can copy into an existing array or create the array themselves. Keyword arguments allow you to specify the range to copy.

Both `primitive-array-to-lisp-array` and `lisp-array-to-primitive-array` require the Lisp array element type to match exactly the Java array element type. The corresponding types are:

<table>
<thead>
<tr>
<th>Java array element type</th>
<th>Lisp array element type</th>
</tr>
</thead>
<tbody>
<tr>
<td>:int</td>
<td>(signed-byte 32)</td>
</tr>
<tr>
<td>:long</td>
<td>(signed-byte 64)</td>
</tr>
<tr>
<td>:short</td>
<td>(signed-byte 16)</td>
</tr>
<tr>
<td>:byte</td>
<td>(signed-byte 8)</td>
</tr>
</tbody>
</table>
get-primitive-array-region and set-primitive-array-region take a primitive array and copy part of it to or from a foreign array ("buffer"), which is passed as an FLI pointer.

### 15.5 Initialization of the Java interface

The Java interface is a module which needs to be loaded by calling:

```
(require "java-interface")
```

Before doing any calls from Lisp to Java or from Java to Lisp or creating any Java object from Lisp, the Java interface must be initialized by a call to `init-java-interface`. `init-java-interface` can either connect to an already running Java virtual machine, or load the JVM library and start it. It has various keyword arguments to set global values.

On Android and in dynamic libraries that were delivered with `setup-deliver-dynamic-library-for-java` with true for `init-java` (the default), the system automatically calls `init-java-interface` on startup.

Merely defining callers to Java and proxies does not use Java. Importing classes needs Java to do the expansion, so will require initializing the Java interface. See discussion in “Importing classes” on page 201.

### 15.6 Utilities and administration

Use `object-p` to test whether a Lisp object is a `object` or not.

Use `lisp-java-instance-p` to test whether the argument is an instance of `standard-java-object`.

<table>
<thead>
<tr>
<th>Java array element type</th>
<th>Lisp array element type</th>
</tr>
</thead>
<tbody>
<tr>
<td>:double</td>
<td>double-float</td>
</tr>
<tr>
<td>:single</td>
<td>single-float</td>
</tr>
<tr>
<td>:char</td>
<td>(unsigned-byte 16)</td>
</tr>
<tr>
<td>:boolean</td>
<td>(unsigned-byte 8)</td>
</tr>
</tbody>
</table>

Table 15.4
get-object returns the object for a Java object, nil otherwise, and can be used as predicate to determine whether the argument is a valid Java object. Note that if you have an instance of standard-java-object, get-object may return nil if the slot is not set. ensure-is-object is like get-object, but signals an error if its argument is not a object.

object-class-name can be used to find the Java class raw name of a Java object. object-pretty-class-name makes it "pretty", which matches how it appears in the Java code.

object-string returns a string representing the object the way Java wants to represent it (the result of Object.toString).

object-to-lisp and lisp-to-object can be used to convert between Lisp and Java objects of primitive types, which may sometimes be useful.

find-java-class can be used to find the Java class object for class specification, which normally is the string representing the full class name, but can be also be a keyword for specific primitive types.

object-of-class-p can be used to verify whether a Java object is an instance of a class or any of its subclasses.

reset-java-interface-for-new-jvm eliminates cached Java objects from internal Lisp structures. It is intended to be used if you need to start a JVM, stop it and start again. Currently there is no interface to stop the JVM.

intern-and-export-list, default-name-constructor, record-java-class-lisp-symbol, ensure-lisp-classes-from-tree and ensure-supers-contain-java.lang.object are utility functions that are used by the definition generation code, and appear in the output of the importing interface (write-java-class-definitions-to-file, write-java-class-definitions-to-stream and generate-java-class-definitions). Their purpose is to be used by the importing interface, but if you find them useful you can call them directly.

get-superclass-and-interfaces-tree returns a tree of the superclasses and interfaces of a Java class. It is also used internally by the importing interface.
send-message-to-java-host can be used to send a message (a string) to the Java host. This is especially useful when the Lisp is used inside Java, for example on Android, so Java needs to do the displaying of messages to the user.

The variables *to-java-host-stream* and *to-java-host-stream-no-scroll* are output streams that send anything that is written to them to Java (by calling send-message-to-java-host). They can be used anywhere an output stream is needed to make the output go to the Java host.

The Java interface currently may generate at run time specific Java interface conditions of the types below.

Conditions with names ending *-exception are all subclasses of java-
exception, and correspond to an exception raised while calling Java. java-
exception has two subclasses: java-normal-exception for exceptions that
you may get during normal execution, and java-serious-exception, for
exceptions that indicate the system is broken in some way. java-serious-
exception should never happen, while java-normal-exception may hap-
pen in normal code.

The other conditions correspond to errors which are detected inside Lisp.

The java-exception class has three readers, java-exception-string,
java-exception-java-backtrace and java-exception-exception-name,
which you can use when handling the condition. The macros catching-
java-exceptions and catching-exceptions-bind can be used to catch
Java exceptions instead of signaling an error. Your code can then access the
Java exception directly.

java-interface-error
Superclass of the *-error conditions.

java-definition-error
Superclass of java-class-error and java-method-
error.

java-class-error
Class not found.

java-method-error
Method not found.
java-field-error
Field not found, or was defined with the wrong *static-p* value.

java-field-setting-error
Setting a field failed, either because it is final or an unacceptable value was supplied.

call-java-method-error
call-java-method failed to find the method.

create-java-object-error
create-java-object failed to find constructors.

java-array-error
Superclass of all array errors.

java-out-of-bounds-error
Bad index passed to jvref or jaref, or bad *start* and *end* passed to other functions accessing arrays.

java-storing-wrong-type-error
Trying to store value of wrong type into a Java array.

java-exception
Superclass of the *-exception* conditions.

java-normal-exception
Superclass of normal exceptions.

java-serious-exception
Superclass of serious exceptions.

Normal exceptions:

field-exception
Superclass of field exceptions.

field-access-exception
15.7  Loading a LispWorks dynamic library into Java

When a LispWorks application is delivered as a dynamic library and is loaded by Java, the Java interface must be initialized at some point to make it possible to use it for interfacing with Java. Lisp code which does not interact with Java will work without initializing Java. This includes the initialization function (the first argument to \texttt{deliver}), which can do any required Lisp initialization. However, calls to Java and accepting calls from Java require initialization of the Java interface.

The function \texttt{setup-deliver-dynamic-library-for-java} is used to set this up. In the simple case, you just call \texttt{setup-deliver-dynamic-library-for-java} without any arguments, and then call \texttt{deliver}. When the resulting delivered Lisp image is loaded into Java, Lisp is initialized as usual, and then \texttt{init-java-interface} is automatically called with the host's Java virtual machine.

\texttt{setup-deliver-dynamic-library-for-java} forces the delivered image to be saved as a dynamic library, which when loaded by Java (normally by \texttt{System.loadLibrary} or \texttt{System.load}) receives the host's Java virtual machine, initializes the Java interface (unless \texttt{init-java} is passed as \texttt{nil}) and then calls its \texttt{function} argument if it is non-nil.

The deliver startup function (the first argument to \texttt{deliver}) is called before Java is initialized, so any code that needs to run before initializing the Java interface should be in this function.
By default the initialization is done synchronously, that is by the time that the Java method that loads the LispWorks delivered library returns, LispWorks has finished initializing and is ready to receive calls from Java and other foreign calls. As a result, the loading code on the Java side will hang until the initialization finishes. `setup-deliver-dynamic-library-for-java` can be told to make initialization asynchronous, that is the loading method just starts the initialization and returns immediately. Calls from Java into Lisp that occur before Lisp is ready will wait until Lisp is ready, and you can check if Lisp is ready by using the Java method `com.lispworks.LispCalls.waitForInitialization`.

`setup-deliver-dynamic-library-for-java` works by internally defining a foreign-callable for `JNI_OnLoad` and exporting it. You must not define this yourself when using `setup-deliver-dynamic-library-for-java`.

If you have your own C code that uses the JNI, you can pass the JVM to Lisp yourself via a foreign-callable (or return it from a lisp-to-foreign call), and call `init-java-interface` with it. In this case, you must not use `setup-deliver-dynamic-library-for-java`.

The function `get-host-java-virtual-machine` can be used to get the Java virtual machine that was passed from Java to the internally defined `JNI_OnLoad`, and can be used as a predicate to test if `JNI_OnLoad` was called. Thus you can create a dynamic library that may be loaded by Java or by a conventional mechanism, and use `get-host-java-virtual-machine` to distinguish between these two situations.

There is a minimal example of delivering LispWorks for Java in:

```
(example-edit-file "java/lisp-as-dll/README.txt")
```

### 15.8 CLOS partial integration

The integration of CLOS is mainly the fact that the functions that take a `job-ject` also accept a CLOS instances of the class `standard-java-object`, which has a slot containing the `job-ject` to use. That includes arguments to Java callers, Java arrays in the array interface, and return values from Lisp to Java. However, values that come from Java to Lisp (return values of caller, arguments in Java to Lisp calls), are always a `job-ject` or primitives.
You can create a subclass of `standard-java-object` either by the usual way of including it (or a subclass of it) in the superclasses of your class, or by using the keywords arguments to importing functions and macro. To be able to construct a `jobject` for a class without the constructor, the `class-name` must be passed to `define-java-constructor`. This is done automatically by the importing functions.

When defining the class using the importing function, you can force it to create the complete hierarchy of superclasses to match all Java superclasses and implemented interfaces. This creates overhead and is not necessarily useful, but in some circumstances it may be what you need. You can also force the hierarchy explicitly by using `ensure-lisp-classes-from-tree`.

The `jobject` in an instance of `standard-java-object` can be read and written by the accessor `java-instance-jobject`. Alternatively you can call `create-instance-jobject` or `create-instance-jobject-list` to create the `jobject` for a given instance.

A simple interface for making an instance and its `jobject` together is `make-java-instance`, but this does not provide a way to pass arguments to `make-instance`. The initarg `:construct` to `make-instance` on a subclass of `standard-java-object` can be used to make the instance and the `jobject`. Note, however, that the `jobject` is created in the `cl:initialize-instance` method of `standard-java-object`, which may or may not be called before your `cl:initialize-instance` methods (depending on the order of the superclasses). To ensure that the `jobject` is created after the CLOS instance initialization is complete, do not pass the `:construct` initarg, and instead call `create-instance-jobject` or `create-instance-jobject-list` afterwards.

The argument to `create-instance-jobject-list` and to the `:construct` initarg is either a list of arguments to the constructor, or `t`, which means use the default arguments list. The default arguments list is created by calling `default-constructor-arguments` on the instance. The default method returns `nil`, which is good enough for some Java classes, but not all. Note that if you pass `:construct` to `make-instance`, `default-constructor-arguments` will be called on the instance before all the `cl:initialize-instance` methods have been called, which may be a problem if it depends on some values that may be put in by other `cl:initialize-instance` methods. To avoid
this issue use `create-instance-jobobject-list` with `t` on the result of `make-instance`.

If you have a `jobobject`, and there is a CLOS class defined for its Java class, you can create a CLOS instance for it using `create-instance-from-jobobject`. `create-instance-from-jobobject` finds the class using the record that is created by `record-java-class-lisp-symbol`. The call to `record-java-class-lisp-symbol` is done automatically by the importing interface, but you can also call it directly.

### 15.9 Java interface performance issues

Both Java and Lisp do memory management on their objects, which causes the interface between them to be problematic. The result is that calls between Java and Lisp are more expensive than calls from and to C, and that keeping a pointer to a Java object (`jobobjects`) in Lisp adds overhead for both sides.

In general, code that needs to be efficient should not make calls between Lisp and Java. For interactive response on a mobile device, as a rough guide, if you have more than 100 calls between Lisp and Java per user gesture, you should reduce the number of such calls, or move the processing to another thread, so that the GUI is still responsive.

Keeping pointers to a Java object in Lisp (`jobobjects`) creates an overhead both for Lisp (which needs to maintain a record so it can tell Java when it is free), and for Java. It is therefore a bad idea to keep large number of pointers to `jobobject` in Lisp. As a rough guide, when you reach 100 objects you should consider changing the interface.

Accessing the first dimension of an array can be done much more efficiently by the multiple access functions than the single element accessors. When accessing a multi-dimensional array, accessing more than one element in a sub-array can be done much more efficiently by getting the sub-array and accessing it instead of accessing via the top array.

If you pass to Lisp an array of Objects where Lisp goes through many of them and just reads one or two values, it is probably faster to put these values into a primitive array or string and pass this to Java instead. This avoids the creation of a `jobobject` and call(s) into Java for each object, which would be much more
expensive than the allocation of a primitive array and filling it in Java. The same is true in the other way.

If you need to pass a very large (megabytes) `Array` or `String` between Java and Lisp, it may be better to write it to a file and pass the filename.
Java interface
Android interface

To use LispWorks for Android Runtime, you need to have at least a minimal Android project written in Java, to load and initialize LispWorks. CAPI is not supported on Android, so any GUI part will need to be written in Java too.

To use the Android interface you need to deliver your application by the special image lispworks-7-1-0-arm-linux-android. This image does not contain the GUI part of LispWorks, but contains all the non-GUI parts.

This special image is an ARM image, and must be run on ARM architecture. That can be either an ARM machine, or an ARM emulator. To deliver a LispWorks for Android Runtime image using the QEMU emulator, you run the special image using the via the shell script examples/android/run-lw-android.sh

The Android interface relies on the Java interface, which is already loaded into the special image. You will typically also use the Java interface in your own code to make calls to Java methods, and define Lisp proxies that can be used inside Java, though in principle the whole interface may be done via direct calls from Java into Lisp, without using the Java interface explicitly.

The interface for Android includes the following:

- The function deliver-to-android-project, which is the function that you use to deliver LispWorks code for Android. The files delivered are a
dynamic library and a Lisp heap, which can then be loaded by and ini-
tialized by the Java com.lispworks.Manager.init method. By default
it delivers the library and heap directly into the directory structure of an
Android project.

- A JAR file containing a few classes in the com.lispworks package to
  support the Java/Lisp interface. This includes these classes:

  com.lispworks.Manager
  Defines the method com.lispworks.Manager.init to load and initial-
  ize LispWorks, error reporting interface and some basic utilities.

  com.lispworks.LispCalls
  Defines direct callers into Lisp, and support for Lisp proxies (which are
  Java proxies that call Lisp functions). LispCalls is really part of the gen-
  eral LispWorks Java Interface.

  com.lispworks.BugFormLogsList and com.lispworks.BugForm-
  Viewer
  Two activities to help display errors during development.

- A few Android-specific interface functions:

  android-funcall-in-main-thread
  android-funcall-in-main-thread-list
  android-get-current-activity
  android-main-thread-p

16.1 Delivering for Android

To use LispWorks in an Android project, the Android project needs three extra
files:

1) lispworks.jar

This defines the support classes in the Java package com.lispworks. This file
is part of the LispWorks distribution, and can be found in the etc directory in
the LispWorks distribution:

(lispworks-file "etc/lispworks.jar")
You must add this file to your project. On Eclipse, it should be added to the build path, though you can also just copy it to the \texttt{libs} directory. On Android Studio 0.4.6 with the default configuration you can put it in \texttt{app/libs}.

2,3) A LispWorks heap and dynamic library. These two files are generated by \texttt{deliver-to-android-project}. Both have the same base name (default "LispWorks"). The library has \texttt{"lib"} before the base name and \texttt{".so"} after it. The heap file name is the library name with the string \texttt{".lwheap"} appended. The heap file needs to be in the \texttt{"assets"} sub-directory of the project, while the dynamic library needs to be in \texttt{armeabi-v7a} sub-directory of the jni libs directory of the project. For Eclipse, the jni libs directory is \texttt{libs} at the top level, and \texttt{assets} is at the top too, so the files are by default in:

\begin{verbatim}
assets/LispWorks.so.lwheap
libs/armeabi-v7a/LispWorks.so
\end{verbatim}

In Android Studio 0.4.6, the jni libs directory is \texttt{jniLibs}, and both it and the \texttt{assets} directory are in the "main" directory, so by default the files are in:

\begin{verbatim}
app/src/main/assets/LispWorks.so.lwheap
app/src/main/jniLibs/armeabi-v7a/LispWorks.so
\end{verbatim}

By default, \texttt{deliver-to-android-project} puts these files in these places, but note that on Eclipse if you have native libraries the default does not work. See full discussion in the entry for \texttt{deliver-to-android-project}.

The \texttt{lispworks.jar} file is required so that your Java IDE knows about classes in \texttt{com.lispworks}, so you need it while working on the Java code that interfaces with Lisp. The other two files are needed only when you actually build the project.

Once these three files are in place, the Android project can be built and installed like any Android project. To use LispWorks, the method \texttt{com.lispworks.Manager.init} must be called to initialize LispWorks. If \texttt{library-name} was passed to \texttt{deliver-to-android-project}, then \texttt{com.lispworks.Manager.init} must be called with a matching name, otherwise the default "LispWorks" is used. \texttt{com.lispworks.Manager.init} can be called at any point during the lifetime of the Android app.

\texttt{com.lispworks.Manager.init} is asynchronous, in other words by the time it returns Lisp is not ready yet. \texttt{com.lispworks.Manager.init} optionally
takes a Runnable argument, which is called when LispWorks is ready. Alternatively the method \texttt{com.lispworks.Manager.status} can be used to determine when LispWorks is ready. See the entry for \texttt{com.lispworks.Manager.init} for more details.

\texttt{com.lispworks.Manager.init} loads LispWorks and initializes it. Apart from standard initialization and starting multiprocessing, the startup function also initializes the Java interface using \texttt{init-java-interface}, passing it the appropriate arguments. That includes passing the keyword \texttt{:report-error-to-java-host}, which makes the function \texttt{report-error-to-java-host} invoke the user Java error reporters, and the keyword \texttt{:send-message-to-java-host} which makes the function \texttt{send-message-to-java-host} call the Java method \texttt{addMessage}. See Chapter 43, “Android Java classes and methods” for the details.

The startup functions also set up a global “last chance” internal debugger hook, which is invoked once the debugger actually gets called (after any hooks you set up like error handlers, debugger wrappers and \texttt{cl:*debugger-hook*}). The hook reports the error to the Java host (that is, invokes the user error reporters) and calls \texttt{cl:abort}. If you did not define a \texttt{cl:abort} restart, that will cause the current process to die, unless it is inside a call from Java, where it will cause this call to return. The return value is a zero of the correct type (see in “Direct calls” on page 206 and “Using proxies” on page 207).

Once initialization finished, if a function was passed to \texttt{deliver-to-android-project} as its \texttt{function} argument, it is invoked asynchronously, and then the \texttt{Runnable} which you passed to \texttt{com.lispworks.Manager.init} (if any) is invoked. From this point onwards, Lisp is ready to receive calls from Java, and can make calls into Java.

On Android when doing GUI operations it is essential to do them from the GUI thread, which is the main thread on Android. The functions \texttt{android-funcall-in-main-thread} and \texttt{android-funcall-in-main-thread-list} can be used to invoke a Lisp function on the main thread. To facilitate testing, these functions are also available on non-Android ports.

There is no proper debugger on Android itself, so it is important to ensure your code is working before delivering.
16.2 Directories on Android

On Android the temp directory that is used by default by open-temp-file and similar functions is the cacheDir of the application context. In principle the system can remove files from this directory when it needs disk space. The documentation for Android says that you should not rely too much on that, and avoid accumulating files in this directory.

LispWorks puts files with names starting with "lw" in this directory, so your code should avoid creating filenames starting with "lw".

get-folder-path can be used to find useful directories. :appdata for private directory, :documents for "user homedir" and :common-appdata for the external directory are the most useful keywords to pass. On Android get-folder-path can also be used to access the standard Android directories like the music and movies directories.

The function cl:user-homedir-pathname on Android returns the result of:

(sys:get-folder-path :documents)

16.3 The Othello demo for Android

The Othello demo is a simple Android app showing the basics of using LispWorks for Android Runtime. It is a full Android project that can be imported into an Android development environment, for example Eclipse with ADT and Android Studio.

The application plays the Othello game as an example of an application. When delivering "with Lisp" (see “Delivering LispWorks to the project” below), it also allows the user to type and evaluate Lisp forms. This is useful during development.

To try the demo, you need to do these steps:

1. Create an Android project containing the demo Android code.
2. Deliver the LispWorks application to it.
3. Build and install the application and run it.

These steps are described in detail in the following sections.
16.3.1 Creating an Android project

The Android project code is in ./OthelloDemo/, which is the directory examples/android/OthelloDemo inside the LispWorks distribution. You need to make a project with this code.

The following sub-sections provide the detail for Eclipse and Android studio projects respectively.

16.3.1.1 Othello project in Eclipse (with ADT)

This procedure was checked with ADT build: v22.6.2:

1. You need to have a workspace (corresponding to a directory), which may be completely empty.
2. Choose File > Import from the menubar to raise the Import dialog.
3. In the Import dialog expand Android, select Existing Android Code Into Workspace and click Next.
4. You should be in the Import Projects tab. Click Browse....
5. Navigate to the examples/android/OthelloDemo directory inside the LispWorks distribution directory, select it and press Ok.
6. Click Select All to ensure that the project is selected.
7. Select the Copy projects into workspace check button (below the list).
8. Click Finish.

The new project name defaults to "LispWorksRuntimeDemo", and this, relative to the workspace directory, is the project path that you will need to set the *project-path* to (below). You can change that before clicking Finish.

Note: if you get errors when trying to build, try:

1. Check that the Project build target is 3.0 or higher. Do: Project > Preferences, click Android, and it shows the list of targets available. Select the highest.
2. Try "cleaning": Project > Clean.
16.3 The Othello demo for Android

16.3.1.2 Othello project in Android studio

This procedure was checked with Android studio 0.4.6:

1. Inside Android studio, select **Import project...** either from the **Welcome to Android studio** dialog or from the **File** menu. That raises the **Select Gradle Project Import** dialog.

2. In the **Select Gradle Project Import** dialog, select the **examples/android/OthelloDemo** directory in the LispWorks distribution and press **OK**. That should raise a dialog called **Import Project from ADT (Eclipse Android)**.

3. Select a directory to put the project in and press **OK**. This is the “project path” that you will need to set the **project-path** variable to (below).

4. Leave all the default settings in the next page and press **Finish**.

16.3.2 Delivering LispWorks to the project

To deliver LispWorks, copy one of the build script files **deliver-android-othello.lisp** or **deliver-android-othello-with-lisp.lisp** from the **examples/android** directory in the LispWorks distribution. In the copied file change the value of the variable **project-path** to point to the directory of the project that you created above. For example:

```
(defvar *project-path* "~/my-workspace/LispWorksRuntimeDemo/")
```

You will then use your edited copy of the build script as the **-build** command line argument to LispWorks.

The Android delivery image is called **lispworks-7-1-0-arm-linux-android**. This must be run on an ARM architecture, currently that means ARM Linux or an emulator. To run this image under the QEMU emulator, use the script **examples/android/run-lw-android.sh**.

```
run-lw-android.sh -build /path/to/deliver-android-othello.lisp
```

See **deliver-to-android-project** for details.

If you cannot access the directories that you use in your project from the ARM machine:
1. Change *project-path* to some temporary directory, and add :no-sub-dir t to the call to deliver-to-android-project. For example, if you use deliver-android-othello.lisp, the call should be:

(deliver-to-android-project nil *project-path* 5 :no-sub-dir t)

This will create two files in the temporary directory called libLisp-Works.so and LispWorks.so.lwheap.

2. Copy libLispWorks.so to the directory libs/armeabi-v7a inside your project, and copy LispWorks.so.lwheap to the assets directory inside your project (assuming you are using Eclipse/ADT).

The two deliver scripts, deliver-android-othello.lisp ("without Lisp") and deliver-android-othello-with-lisp.lisp ("with Lisp") differ in what the application contains. The "with Lisp" script keeps a lot of Lisp in the application, and hence allows evaluation of Lisp forms when running on Android. The "without Lisp" script delivers at the maximum delivery level, and hence cannot evaluate forms. As a result the runtime image is much smaller.

16.3.3 Running the application

Once you have the project with the LispWorks files, you can build, install it on the device and run it as any other Android project. When it runs, it first shows a splash screen (the LispWorks splash screen image) and then the first screen displays an Othello board, where you can play against the computer (you play black), by touching the square where you want to add your piece.

The display has two elements in addition to the board:

- A small text view which displays the status of the game, and
- A checkbox Computer Plays, which controls whether the computer plays. When the computer does not play, the board is set for two players.

It also has a menu (which maybe partly displayed on the action bar), with these items:

- Restart, Restart the game.
16.3 The Othello demo for Android

**Undo**  
Undo the last move. You can undo repeatedly to the beginning of the game.  
When the computer plays, each undo undoes to the state before your last move.  
When the computer does not play, it undoes one move.

When delivering "with Lisp" the menu also has these items:

- **Lisp Panel**  
  Takes you to the Lisp Panel screen, which allows you to evaluate Lisp forms. See below in the description of the Lisp Panel.

- **Command history**  
  Takes you a list of the forms that that you evaluated. It is initialized by a few demo forms. See below about the History list.

- **Othello Server**  
  Raises a submenu with three items: **Java server, full proxy** and **lazy proxy**. Switching between these changes the mechanism by which Java calls into Lisp. The behavior of the game is exactly the same, only the output to the **Lisp Panel** or **Output** is different. This feature is for demonstrating different techniques of calling from Java to Lisp. See discussion of the code for details.

When delivering "without Lisp" the menu also has these items:

- **Output**  
  Takes you to the "output" screen.

### 16.3.3.1 The Lisp Panel screen

The Lisp Panel contains a row of buttons, a text view for input, and the bottom is a text view for output. This screen is available only when delivering "with Lisp". When delivering "without Lisp", there is the Output screen instead.

The buttons are:

- **Clear**  
  Clears all the output from the output pane.

- **Evaluate the string**  
  Send the current text in the input pane to Lisp by a direct call to `eval-for-android`.
eval-for-android is defined in (example-edit-file "android/android-othello-user"). It reads the string and evaluates it. If it is successful, it prints to the output pane the form, anything the form printed, and the result(s). If there is an error, it logs the error and prints the error message to the output pane.

**History**

Takes you to another screen which displays a list of the forms that were evaluated. The list is initialized by some forms which demonstrate some features of the multiprocessing on Android. See below in the section “Prepared forms”. Whenever you evaluate a form by pressing **Evaluate the string**, it adds the form to the history in the beginning of it. If the form matches exactly a form which is already in the list, the old item is removed.

In the history list, when you touch an item it is inserted into the input pane, and the application switches to the Lisp Panel. It does not evaluate the form at that point. You can also reach the history list from the menu in the Othello screen.

**Bug form logs**

Invokes com.lispworks.Manager.showBugFormLogs. This shows another screen with a list of the logged errors displaying the error string for each item. Touching an item opens another screen with bug form log of this error.

**Clear logs**

Clears all the bug form logs, including removing the files.

The input pane below the buttons is just a passive text view, in which you can type Lisp forms, and evaluate by touching the **Evaluate the string** button.

The bottom part of the Lisp Panel, in the Output screen when delivering “without Lisp”, is the output pane. It prints the output of evaluation as above. It also prints whenever you touch a square in the Othello board. When the Full or Lazy proxy is used for communication, it also prints this fact.
16.3.3.2 Prepared forms

Initially, the History list contains the forms described below. When using forms, note that evaluating a form moves it to the top of the list. When you should evaluate more than one of these forms in order, you will need to look down the list for each one in turn.

The idea is that you can try these forms, and then modify them to check and perform things that you need to do when debugging your application.

Forms:

1. 
   (mp:ps)
   Shows the Lisp processes. Initially there are at least the idle process and the GUI process which displays as "created by foreign code".

2. 
   (setq *computer-plays-waste-time-in-seconds* 2)
   That causes the computer to pretend that it takes it time to compute a move. When playing against the computer after setting this, you will see that after your move, the display says "Computer to play" for two seconds before it actually plays. Set *computer-plays-waste-time-in-seconds* back to nil to make it behave normally.

3. 
   (defun eval-and-print (form)
      (let ((res (eval form)))
         (lw-ji:send-message-to-java-host
          (princ-to-string res) :reset)))
   Defines a function to be used by the next two forms. Note that it uses send-message-to-java-host to print, which comes in the output and works on any thread. When it is on the current thread it will end up printing before the printing of the evaluation, but on another thread it is random which output comes first.

4. 
   (eval-and-print '(mp:get-current-process))
Use the function defined above to print the process in the current thread. That is the GUI process.

5.

(mp:funcall-async 'eval-and-print '(mp:get-current-process))

Use the function eval-and-print defined above to print the process on which funcall-async executes the function. This will be one of the Background Execute processes.

6.

(progn
  (defun loop-executing-events ()
    (loop
      (let ((event (mp:process-wait-for-event)))
        (lw-ji:format-to-java-host "~%got event ~s" event)
        (let ((res (mp:general-handle-event event)))
          (lw-ji:format-to-java-host "~%Handling got ~s" res))))
    (setq loop-executing-events-process
        (mp:process-run-function "Loop Execute Events" ()
          'loop-executing-events)))

Create a process called "Loop Execute Events" and set loop-executing-events-process to it. The process has a process function loop-executing-events which read events and handles them using process-wait-for-event and format-to-java-host. It prints "got event <event>" and then "handling got <result of handling>". Note the usage of format-to-java-host, which prints to the output pane too (it actually calls send-message-to-java-host).

7.

(mp:process-send loop-executing-events-process '(mp:get-current-process))

Sends to the "Loop Execute Events" process (that started in the previous step) an event, which cause get-current-process to be called, and hence return the process. You should see "got event (MP:GET-CURRENT-PROCESS)" and "Handling got <process name>"
(othello-user-change-a-square 5 2)

Changes square 5 (sixth from the left in the top row) to color 2 (black).
This function is defined in (example-edit-file "android/android-othello-user") and is part of the "interface" that the Lisp Othello code uses to tell Java to change the board.

9.

(mp:process-run-function
"multiplier" ()
#'(lambda()
  (setq *finish-multiply* nil)
  (dotimes (x 100)
    (sleep 1)
    (when *finish-multiply* (return))
    (lw-ji:format-to-java-host
      "\%-d * %-d = %-d\n"
      x x (* x x)))))

Starts a process that performs "a lengthy computation" (simulated by using (sleep 1)) and prints results while doing it. In each "step in the computation" (the cl:dotimes iteration) it prints the square of the iteration number. To stop it, evaluate the next form.

10.

(setq *finish-multiply* t)

Tell the "multiplier" process (see above) to stop.

11.

(mp:process-run-function
"Error"
() #'(lambda () (open "junk;;file::name")))

Starts another process that gets an error (because the argument to cl:open is an illegal pathname). It prints that it got the error, and you can use the Bug form logs button to look at the bug form log.

12.

(raise-alert-dialog
"What do you want to eat?" +
:ok-title "Chicken "
:ok-callback '(raise-alert-dialog "Here is some chicken") +
:cancel-title "Salad "
:cancel-callback '(raise-alert-dialog "We do not have salad")

235
16 Android interface

Raises an alert dialog using `raise-alert-dialog` which is defined in `dialog.lisp`. Note that this works because the `LispPanel` class uses `com.lispworks.Manager.setCurrentActivity` to set the current activity.

13. 

`(raise-a-toast "Bla Bla Bla" :gravity :left)`

Raises an Android "toast" at the middle of the left side, using `raise-a-toast` which is defined in `toast.lisp`.

16.3.4 Lisp interface usage in the Java code

The Othello Demo Java code is in the package `com.lispworks.example.othellodemo`. LispWorks interfaces in Java are all in the package `com.lispworks`. The methods appear in full, to make it is easy to see where there is a call to the LispWorks interface.

16.3.4.1 Class Othello

`Othello` is a subclass of `Activity` that displays the screen with the Othello board. The display is all in standard Java. The board is made of a grid of 64 `ImageView` panes, each one displaying one of three images (blank, white, black). Each view has an `OnClickListener(SquareListener)` that remembers its index and passes it when clicked.

The Java code does not know anything about the game that is being played, and does not keep a record of the state of the game. That is all done in Lisp.

The Java code processes user gestures concerning the game (touching the board, and touching any of the buttons and items `Computer plays, undo move, restart`) by calling methods on an object that implements the nested interface `OthelloServer`, which is kept in `mOthelloServer`. The object can be either a Lisp proxy, or of the nested class `JavaOthelloServer`. All of these objects do exactly the same thing (calling the Lisp functions defined in `(example-edit-file "misc/othello")`), and the purpose of having all these options is to demonstrate different techniques to call into Lisp. There is also a nested class `ErrorOthelloServer` in case LispWorks does not work, which displays the error. `mOthelloServer` is set by the method `setupServer`.
The nested class `JavaOthelloServer` is plain Java with methods that call into Lisp using the “Direct calls” interface (`com.lispworks.LispCalls.callIntV` and `com.lispworks.LispCalls.callVoidV`). This has the advantage that on the Lisp side all you have to do is to ensure that the functions are not shaken, which you can do with `hcl:deliver-keep-symbols` (see the *LispWorks Delivery User Guide*). It has the disadvantage that you hardwire Lisp function names in Java (though the names can be variables too).

The other two possible implementations of the `OthelloServer` are Lisp proxies which are defined in Lisp (in `examples/android/android-othello-user.lisp`). See the discussion of the Lisp code for more details. The code in `setupServer` demonstrates two techniques of using the proxy definitions: either calling a Lisp function that makes a proxy (using `com.lispworks.LispCalls.callObjectV` to call `create-lisp-othello-server`), or using `com.lispworks.LispCalls.createLispProxy` with the name of the proxy definition (`lisp-othello-server-lazy`) to create a proxy.

To actually respond to moves, the `Othello` class exports 3 methods ("updateState", "signalBadMove" and "change") which are called directly from Lisp to change the board and the status text.

When an Othello instance is created, it calls `setupAndInit` to do anything with Lisp (mainly call `mOthelloServer.init`). Before doing anything that may interact with Lisp, it checks the status of Lisp using `com.lispworks.Manager.status`. If Lisp is not ready and there was no error, it calls `com.lispworks.Manager.init` to initialize LispWorks, passing it a `Runnable` that with call `setupAndInit` again to actually do the initialization. In the Demo the Lisp side will already be initialized, because it is done by the *LispWorksRuntimeDemo* activity, but the Othello class avoids relying on it.

When LispWorks is ready, `setupAndInit` sets up the server by calling `setUpServer` and initializes the game by calling `mOthelloServer.init`.

If there is an error, `setupAndInit` gets the error details using `com.lispworks.Manager.mInitErrorString` and `com.lispworks.Manager.init_result_code` and adds a message, set `mOthelloServer` to `ErrorOthelloServer`, and then shows the Lisp Panel which will be displaying the error.
There is also an `onCreateOptionsMenu` method which checks whether Lisp is working and can evaluate forms (using `LispPanel.canEvaluate`), and accordingly decides which menu to use.

### 16.3.4.2 Class LispPanel

LispPanel is a subclass of `Activity` that displays the Lisp panel, or just the output when delivering "without Lisp" (see “Delivering LispWorks to the project” on page 229).

The main purpose of the Lisp Panel is to evaluate Lisp forms, which it does by calling the Lisp function `eval-for-android` using `com.lispworks.LispCalls.callIntV`. That can work only if `eval-for-android` is defined, so LispPanel has a method `canEvaluate` that works by checking if `eval-for-android` is defined using `com.lispworks.LispCalls.checkLispSymbol`. If `eval-for-android` is `fbound`, LispPanel displays in full, otherwise it shows only output `TextView`. LispPanel is also responsible for displaying messages in its output `TextView`. To achieve that, it uses `com.lispworks.Manager.setTextView`. Once it sets the `TextView`, all calls to `com.lispworks.Manager.addMessage` and calls to the Lisp functions `send-message-to-java-host` and `format-to-java-host` put their output in this `TextView`.

Other usage of the `com.lispworks` package in LispPanel are:

- `com.lispworks.Manager.setErrorReporter` to set an error reporter. Since the Lisp application does not set `cl:*debugger-hook*`, uncaught errors will end up calling this reporter.

- Calls to `com.lispworks.Manager.showBugFormLogs` to show bug form logs, and `com.lispworks.Manager.clearBugFormLogs` to clear them.

- Calls to `com.lispworks.Manager.setCurrentActivity` in `onResume` and `onPause` to allow Lisp code to raise dialogs when LispPanel is visible. This is needed to allow the `raise-alert-dialog` form to work.

### 16.3.4.3 Class MyApplication

MyApplication is not actually used in the demo. It is a demonstration of how to initialize LispWorks when the application starts, by calling `com.lisp-
works.Manager.init in the onCreate of the application. The demo itself does not use this mechanism. Instead the SplashScreen activity does it, and the Othello activity also checks using com.lispworks.Manager.status, and if LispWorks needs initializing does it.

16.3.4.4 Class LispWorksRuntimeDemo

Display a splash screen and initialize the Lisp side, by checking com.lispworks.Manager.status and using com.lispworks.Manager.init if needed. The purpose of this class is just to give an example of displaying a splash screen while initializing Lisp. It is not really needed, because the Othello class checks too (in setupAndInit). On Eclipse the name of this class is the default project name.

16.3.4.5 Class History

A simple class to display Lisp forms. Does not do anything related to Lisp.

16.3.4.6 Class SquareLayout

A simple class to make a square layout for displaying the Othello board. Does not do anything related to Lisp.

16.3.5 Java and Android interface in the Lisp code

The file

(example-edit-file "misc/othello")

is a generic implementation of the playing Othello part, and has nothing to do with Java or Android.

The Lisp code that interacts with Java and Android to play Othello and evaluate the forms is in

(example-edit-file "android/android-othello-user")

The Java callers to update the game are defined by a define-java-caller form. All these methods need to be called on the GUI thread (because they interact with GUI elements), so the actual functions that are called from the
Othello code are defined to call the Java callers using `android-funcall-in-main-thread`.

The function `eval-for-android` is what the Java code uses to evaluate Lisp forms. The function has no Java-specific features, but it has error handling and binding of some of the top-level variables like `cl:*` to make it more usable in repeated calls from "outside".

The code also defines two proxy definitions that implement the `Othello.OthelloServer` interface which responds to user gestures. To demonstrate the various features of proxies, there are two definitions which achieve exactly the same thing. The full proxy definition (`lisp-othello-server-full`) specifies functions for all the methods that the interface defines. The lazy (programmer) proxy definition does not define any method. Instead it has a default function that decides what to do based on the method name.

The two files

```
(example-edit-file "android/dialog")
```

and

```
(example-edit-file "android/toast")
```

define the functions `raise-alert-dialog` and `raise-a-toast` respectively, to demonstrate using Android code directly from Lisp. See the comments in these files.
To build an application using LispWorks for iOS Runtime, you need an Xcode project to implement the main function of the application. CAPI is not currently supported on iOS, so any GUI part will need to be written in Objective-C using Xcode too.

### 17.1 Delivering for iOS

The Lisp part of the application needs to be delivered using one of four special images:

- **lispworks-7-1-0-x86-darwin-ios** and **lispworks-7-0-0-amd64-darwin-ios** which build runtimes for the iOS Simulator, 32-bit and 64-bit, running on Mac OS X, or

- **lispworks-7-1-0-arm-linux-ios** and **lispworks-7-0-0-arm64-linux-ios** which build runtimes for the real iOS device, 32-bit and 64-bit. To do this, you run the QEMU emulator on Mac OS X and tell it to run the lispworks-7-1-0-arm-linux-ios or lispworks-7-0-0-arm64-linux-ios image.

There is an example script examples/ios/run-lw-ios.sh which can be used to invoke all of these images in sequence with the same command line arguments.
These images do not contain the GUI part of LispWorks, but do contain all the non-GUI parts.

There are no iOS-specific Lisp functions required to build the Lisp part of an application: you use `deliver` in the normal way. The only difference compared to a desktop application is that the file passed to `deliver` should have a ".o" extension and the generated file will be an iOS object file that must be linked with the other parts of the application using Xcode. Generation of shared libraries is not supported (this is a limitation of iOS).

To include the delivered object file in an Xcode project, add the file to the project using the `File > Add files to <project>...` menu item. Xcode should detect this as a file of type "Object Code".

If you want to run the same Xcode project with both the iOS Simulator and a real device or build for 32-bit and 64-bit devices then you can conditionalize the filename passed to `deliver` to create four different files. For example,

```
(deliver 'init-othello-server
 (merge-pathnames
  (format nil "OthelloDemo/OthelloServer-~A.o"
     #+arm "armv7"
     #+arm64 "arm64")
     #+x86 "i386")
     #+x86-64 "x86_64")
 *project-path*)
0
:keep-symbols '(othello-server))
```

Then add `OthelloServer-armv7.o`, `OthelloServer-arm64.o`, `OthelloServer-i386.o` and `OthelloServer-x86_64.o` to the Xcode project. Building the project will result in warnings such as

```
ld: warning: ignoring file
/Users/developer/Documents/OthelloDemo/OthelloDemo/OthelloServer-armv7.o, file
was built for armv7 which is not the architecture being linked
(i386):
/Users/developer/Documents/OthelloDemo/OthelloDemo/OthelloServer-armv7.o
```

These warnings can be ignored.
17.2 Initializing LispWorks

In order to use Lisp code within an application built using Xcode, the main function of the application must call \texttt{LispWorksInitialize}. For example, \texttt{main} might be implemented like this:

```c
#import "LispWorks.h"

int main(int argc, char *argv[]) {
    if (!LispWorksInitialize(argc, argv)) abort();

    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, 
            NSStringFromClass([OthelloAppDelegate class]));
    }
}
```

\texttt{LispWorksInitialize} is automatically included in the object file generated by \texttt{deliver}. The file \texttt{LispWorks.h} can be found in the \texttt{examples/ios/OthelloDemo/OthelloDemo/} directory of the LispWorks installation and should be copied into the Xcode project.

17.3 Using Objective-C from Lisp

LispWorks calls the function \texttt{objc:ensure-objc-initialized} when it starts the iOS runtime, so there is no need for you to call it.

For other details, see the \textit{LispWorks Objective-C and Cocoa Interface User Guide and Reference Manual}.

17.4 Limitations of the iOS Runtime

There are some limitations that iOS imposes that affect the LispWorks for iOS Runtime.

- Compiled code cannot be generated, so \texttt{cl:compile} cannot be called at run time.
- Shared libraries cannot be loaded dynamically, so \texttt{fli:register-module} cannot be used. Instead, add references to any required frameworks in the Xcode project.
17.5 The Othello demo for iOS

The Othello demo is a simple iOS app showing the basics of using LispWorks for iOS Runtime. It contains an Xcode project to run the GUI and some Lisp source code to play the game.

To try the demonstration, see the file

(example-edit-file "ios/README.txt")

17.5.1 Notes about the Xcode project

The Xcode project in examples/ios/OthelloDemo/ has a standard layout, with the class OthelloAppDelegate defined in

(example-edit-file "ios/OthelloDemo/OthelloDemo/OthelloAppDelegate.m")

implementing the UIApplicationDelegate protocol.

The file

(example-edit-file "ios/OthelloDemo/OthelloDemo/main.m")

initializes LispWorks by calling LispWorksInitialize and then runs the application main loop using the OthelloAppDelegate.

The application has two storyboards (MainStoryboard_iPhone and MainStoryboard_iPad) which display a Tab Bar allowing you to switch between an Othello game and a Lisp evaluation pane.

17.5.2 The Othello game

The Othello game is displayed by the Othello scene and contains an Othello board (the boardView outlet), a few buttons on a toolbar and a label showing the state of the game (the stateView outlet).

The scene is controlled by the class OthelloViewController, defined in

(example-edit-file "ios/OthelloDemo/OthelloDemo/OthelloViewController.m")

The 64 tiles on the board are represented by UIImageView objects, created dynamically in the viewDidLoad method. The contents of the tiles are the
images "empty", "white" and "black" which are loaded from the Images.xcassets asset catalog. The viewDidLoad method also creates an instance of the class OthelloServer, which is implemented in Lisp (see “Notes about the Lisp code” on page 246).

The tiles in the board are dynamically positioned by the viewDidLoad method.

The action methods restartOthello: and undoMove: are connected to the toolbar buttons in the storyboards and call into the Lisp code to update the game.

The action method playUISquare: is triggered when the user touches a square on the board (see viewDidLoad) and calls into the Lisp code to play that square.

The methods changeOthelloSquare:, updateStateString: and signalBadMove are called by the Lisp code to modify the GUI.

17.5.3 The Lisp evaluation pane

The Lisp evaluation pane is displayed by the Lisp Panel scene and contains a text field for entering a Lisp form (the formInputView outlet), a text field to display the evaluation results (the textOutputView outlet) and toolbar.

The scene is controlled by the class LispPanelViewController, defined in

(example-edit-file "ios/OthelloDemo/OthelloDemo/
LispPanelViewController.m")

The action methods evaluate:, clearTextOutput: and showHistory: are connected to the toolbar buttons in the storyboards.

The History button pops up a history of the forms entered so far. This is displayed by the History Table scene controlled by HistoryTableViewController, defined in

(example-edit-file "ios/OthelloDemo/OthelloDemo/
HistoryTableViewController.m")

and communicates back to the LispPanelViewController using the HistoryTableViewControllerDelegate protocol.
The keyboardWasShown: and keyboardWillBeHidden: notification methods resize the textOutputView to avoid the on-screen keyboard.

The method appendTextOutputString: is called by Lisp code to update the textOutputView.

### 17.5.4 Notes about the Lisp code

The Lisp code triggered by the GUI is in the file

```plaintext
(example-edit-file "ios/ios-othello-user")
```

and uses the shared Othello logic in

```plaintext
(example-edit-file "misc/othello")
```

The function init-othello-server is the main entry point of the Lisp code and is called when LispWorksInitialize is called from main.m. It initializes the LispWorks Objective-C interface and creates a helper object (lispworks-main-threads-funcalls-object) used by invoke-in-main-thread for making Lisp calls in the main thread of the application.

The Lisp code implements an Objective-C class OthelloServer using the Lisp class othello-server. This class implements the methods initWithViewController for initialization and initOthello, playSquare: and undoMove for the Othello game GUI code to call into Lisp in response to user gestures.

The Lisp code also implements the functions othello-user-change-a-square, othello-user-update-state-string, othello-user-signal-bad-move and othello-user-print-diagnostics-message which the shared Othello logic calls to update the GUI. Most of these functions call methods on the OthelloViewController object, taking care to invoke them in the main (GUI) thread of the application. This thread switching is needed because the Othello logic plays the game in a background thread to avoid hanging the GUI while considering its move (see perform-computer-play in examples/misc/othello.lisp).

Finally, the Lisp code implements an Objective-C class LispPanelServer using the Lisp class lisp-panel-server, with a method evaluate: to evaluate a Lisp form. This evaluate: method is called by the evaluate: action method in LispPanelViewController.
17.6 The Mobile GC

The Mobile GC is a 64-bit GC that is written to run on 64-bit iOS (we are also considering using it for 64-bit Android). When LispWorks is delivered for 64-bit iOS, the "saved image" (the code in the object file that delivery creates) switches automatically to use the Mobile GC. Thus normally you do not need to know anything about the Mobile GC, and if your application has moderate memory requirements (a few tens of megabytes of live allocation at run time), then there is a good chance that you do not need to do anything about memory management.

However, it is useful in general to create a log file of the activity of the application (at least during development), and to include the output of periodic calls to (room) or (room nil) in this log, which will give you some idea of the memory behavior of your application. In particular, if the "Total allocated" that is reported in the last line of (room) (which is the only line in (room nil)) approaches 100 MB then you may want to look at memory management. Note that is only for a mobile device: Desktop applications can grow much larger without a problem.

The main problem that you may encounter in the Mobile GC are GCs of generation 2 causing noticeable delays (on the order of 1 second, depending on the size of the application and the hardware). Depending on the application, such delays may or may not be an issue, but for most GUI applications you should ensure they do not happen too frequently. In this situation you should consult “Preventing/reducing GC of generation 2” on page 159.

You may also consider implementing a response to low memory warning from the OS. All applications can benefit a little by calling reduce-memory with nil or 0 or 1 at that point, but for most applications the benefit is minimal. If most of the live memory in your application contains caches that can be freed and recreated again without much loss in performance, you may be able to get significant improvement by implementing a low memory warning response that clears the caches. Consult “Response to low memory” on page 158 for details.

The output of room and the results of related functions are different when the Mobile GC is used. If you want to understand this output better, consult “Mobile GC technical details” on page 154.
iOS interface
The Metaobject Protocol


There are some discrepancies between LispWorks and AMOP, which are described in this chapter, which also describes some common problems encountered by programmers using the MOP.

18.1 Metaobject features incompatible with AMOP

18.1.1 Instance Structure Protocol

The generic functions implementing slot access are like those described in AMOP, except that each takes a *slot-name* argument rather than a slot definition object, and the primary methods are therefore specialized differently.

For details, see *slot-boundp-using-class*, *slot-value-using-class* and *slot-makunbound-using-class*.
Note: by default, standard slot accessors are optimized to not call `slot-value-using-class`. This can be overridden with the `:optimize-slot-access` class option. See the second definition of `virtual-metaclass` below for an example of the use of this.

`standard-instance-access` is not supported as defined in AMOP. Note that there is an internal function of the same name, but this is not optimal. Also, `funcallable-standard-instance-access` is not supported. An alternative for fast instance access is to use the `:optimize-slot-access` class option.

### 18.1.2 Method Metaobjects

`standard-reader-method`, `standard-accessor-method` and `standard-writer-method` all have a required `:slot-name` initarg, rather than a `:slot-definition` initarg as specified in AMOP.

Compatibility note: in LispWorks 4.3 and previous versions, `accessor-method-slot-definition` was not implemented. This is implemented in the current version.

### 18.1.3 Method Lambdas

LispWorks `make-method-lambda` is not AMOP-compatible. It takes separate `lambda-list` and `body` arguments, and the returned `lambda` form is different to that specified in AMOP (see “Method Functions” on page 250 below).

LispWorks does not support user defined methods for the generic function `make-method-lambda`.

### 18.1.4 Method Functions

LispWorks method functions take the same arguments as the method itself, whereas in AMOP they take a list of arguments and a list of next methods.

### 18.1.5 EQL specializers

`eql-specializer`, `eql-specializer-object` and `intern-eql-specializer` are not implemented.

`eql` specializers in LispWorks are lists.
18.1.6 Generic Function Invocation Protocol

`compute-applicable-methods-using-classes` is not implemented.

`compute-discriminating-function` is implemented and returns the discriminator but:

- It does not use `compute-applicable-methods-using-classes` since LispWorks does not have that function.
- It does not call `compute-applicable-methods`.

Moreover `add-method` does not call `compute-discriminating-function` because this would be inefficient when doing multiple calls to `add-method`. Instead, `compute-discriminating-function` is called when the generic function is called.

18.1.7 Method combinations

`method-combination` objects do not contain the arguments, merely the type. There is a single `method-combination` object per type.

Therefore the value returned by `generic-function-method-combination`, and the default value of the `:method-combination` initarg, and the `:method-combination` argument processed by `ensure-generic-function-using-class` are specific only to the type of the method combination.

Also, `find-method-combination` is not implemented.

18.1.8 Compatible metaclasses

The AMOP defines that the standard primary method for `validate-superclass` should return true if the class of one of the arguments is `standard-class` and the class of the other is `funcallable-standard-class`.

In LispWorks, objects of these metaclasses are not completely compatible, so `validate-superclass` will return false in these cases.

Beware that defining a class that mixes `standard-class` and `funcallable-standard-class` can lead to inconsistencies with the predicate `functionp`, the type `function` and the class `function`.
18.1.9 Inheritance Structure of Metaobject Classes

`funcallable-standard-object` is implemented as defined in AMOP, except that its class precedence list has direct superclasses

```lisp
(funcallable-standard-object)
```

rather than

```lisp
(funcallable-standard-object function)
```

so that LispWorks is compliant with the ANSI Common Lisp rules.

For details, see `funcallable-standard-object`, page 494.

18.2 Metaobject features additional to AMOP

18.2.1 Computing the effective method function

The generic function `compute-effective-method-function-from-classes` is called by LispWorks to compute the effective method function. You can add methods to implement non-standard behavior for your own classes of generic functions.

18.3 Common problems when using the MOP

18.3.1 Inheritance across metaclasses

Usually an inherited class is of the same metaclass as the parent class.

For other kinds of inheritance, you need to define a method on `validate-superclass` which returns true when called with the respective metaclasses. For example:
18.4 Implementation of virtual slots

```
(defclass mclass-1 (standard-class) ()
)
(defclass mclass-2 (standard-class) ()
)
(defclass a () () (:metaclass mclass-1))
(defmethod validate-superclass ((class mclass-2) (superclass mclass-1)) t)
(defclass b (a) () (:metaclass mclass-2))
```

Without the `validate-superclass` method, the last form signals an error because `mclass-1` is an invalid superclass of `mclass-2`.

18.3.2 Accessors not using structure instance protocol

By default, `defclass` creates optimized standard accessors which do not call `slot-value-using-class`.

This optimization is controlled by the `defclass` option `:optimize-slot-access`, which defaults to `t`.

There is an illustration of this effect of `:optimize-slot-access` in the example below.

18.3.3 The MOP in delivered images

Issues with MOP code that occur only in delivered LispWorks images are documented in the section “Delivery and the MOP” in the *LispWorks Delivery User Guide*.

18.4 Implementation of virtual slots

This is an implementation of virtual slots with readers, writers and which also allow access by `slot-value`. 
;;;; ----------------------- Virtual Slots -----------------------
(in-package "CL-USER")

;;;; Metaclass of objects that might contain virtual slots.

(defun virtual-metaclass (standard-class)
  ()
)

;;;; Mixin metaclass for virtual slots and methods to make them
;;;; appear virtual.

defclass virtual-slot-definition
  (standard-slot-definition)
  ((function :initarg :function
              :accessor virtual-slot-definition-function))
)

(defun slot-definition-allocation
  ((slotd virtual-slot-definition))
  :virtual)

(defun (setf slot-definition-allocation)
  (allocation (slotd virtual-slot-definition))
  (unless (eq allocation :virtual)
    (error "Cannot change the allocation of a ~S" 'virtual-direct-slot-definition) allocation)

;;;; Class of direct virtual slots and methods to construct them
;;;; when appropriate.

defclass virtual-direct-slot-definition
  (standard-direct-slot-definition
   virtual-slot-definition)
  ()
)

;;;; Called when the class is being made, to choose the metaclass of
;;;; a given direct slot. It should return the class of slot
;;;; definition required.

defmethod clos:direct-slot-definition-class
  ((class virtual-metaclass) &rest initargs)
  ;; Use virtual-direct-slot-definition if appropriate.
  (if (eq (getf initargs :allocation) :virtual)
      (find-class 'virtual-direct-slot-definition)
18.4 Implementation of virtual slots

(call-next-method))

;; Called when the defclass is expanded, to process a slot option.
;; It should return the new list of slot options, based on
;; already-processed-options.
(defmethod clos:process-a-slot-option
  ((class virtual-metaclass) option value
   already-processed-options slot)
  ;; Handle the :function option by adding it to the
  ;; list of processed options.
  (if (eq option :function)
      (list* :function value already-processed-options)
      (call-next-method)))

;; Class of effective virtual slots and methods to construct
;; them when appropriate.
(defclass virtual-effective-slot-definition
  (standard-effective-slot-definition virtual-slot-definition)
  ()
)

;; Called when the class is being finalized, to choose the
;; metaclass of a given effective slot. It should return the
;; class of slot definition required.
(defmethod clos:effective-slot-definition-class
  ((class virtual-metaclass) &rest initargs)
  ;; Use virtual-effective-slot-definition if appropriate.
  (let ((slot-initargs (getf initargs :initargs)))
    (if (member :virtual-slot slot-initargs)
        (find-class 'virtual-effective-slot-definition)
        (call-next-method)))))

(defmethod clos:compute-effective-slot-definition
  ((class virtual-metaclass)
   name
direct-slot-definitions)
  ;; Copy the function into the effective slot definition
  ;; if appropriate.
  (let ((effective-slotd (call-next-method)))
    (dolist (slotd direct-slot-definitions)
      (when (typep slotd 'virtual-slot-definition)
        (setf (virtual-slot-definition-function effective-slotd)
(virtual-slot-definition-function slotd))
  (return)))
  effective-slotd))

;; Underlying access methods for invoking
;; virtual-slot-definition-function.
(defmethod clos:slot-value-using-class ((class virtual-metaclass) object slot-name)
  (let ((slotd (find slot-name (class-slots class)
                :key 'slot-definition-name)))
    (if (typep slotd 'virtual-slot-definition)
      (funcall (virtual-slot-definition-function slotd)
               :get
               object)
      (call-next-method))))

(defmethod (setf clos:slot-value-using-class) (value (class virtual-metaclass) object slot-name)
  (format t "%-~A" slot-name)
  (let ((slotd (find slot-name (class-slots class)
                 :key 'slot-definition-name)))
    (if (typep slotd 'virtual-slot-definition)
      (funcall (virtual-slot-definition-function slotd)
               :set
               object
               value)
      (call-next-method))))

(defmethod clos:slot-boundp-using-class ((class virtual-metaclass) object slot-name)
  (let ((slotd (find slot-name (class-slots class)
                 :key 'slot-definition-name)))
    (if (typep slotd 'virtual-slot-definition)
      (funcall (virtual-slot-definition-function slotd)
               :is-set
               object)
      (call-next-method))))

(defmethod clos:slot-makunbound-using-class ((class virtual-metaclass) object slot-name)
  (let ((slotd (find slot-name (class-slots class)
                 :key 'slot-definition-name)))
    (if (typep slotd 'virtual-slot-definition)
      (funcall (virtual-slot-definition-function slotd)
               :unset
               object)
      (call-next-method)))))
18.4 Implementation of virtual slots

(defun clos:slot-exists-p-using-class
  ((class virtual-metaclass) object slot-name)
  (or (call-next-method)
      (and (find slot-name (class-slots class)
                :key 'slot-definition-name)
           t)))

;; Example virtual slot which depends on a real slot.
;; Compile this separately after the virtual-metaclass etc.

(defun a-virtual-class-virtual-slot-function
  (key object &optional value)
  (ecase key
    (:get (let ((real-slot (real-slot object)))
            (if (<= 0 real-slot 100)
                (/ real-slot 100.0)
                (slot-unbound (class-of object)
                              object
                              'virtual-slot))))
    (:set (setf (real-slot object) (* value 100))
           value)
    (:is-set (let ((real-slot (real-slot object)))
              (<= real-slot 100)))
    (:unset (setf (real-slot object) -1)))

;; ----------------------- Virtual Slots ---------------------

Compile the code above. Then make an object and access the virtual slot:
CL-USER 1 > (setf object (make-instance 'a-virtual-class))
#<A-VIRTUAL-CLASS 2067B064>

CL-USER 2 > (setf (virtual-slot object) 0.75)

setf slot : VIRTUAL-SLOT
0.75

CL-USER 3 > (virtual-slot object)
0.75

CL-USER 4 > (real-slot object)
75.0

Note that when you call (setf real-slot) there is no output, since
(setf clos:slot-value-using-class) is not called. Compare with
(setf virtual-slot).

CL-USER 5 > (setf (real-slot object) 42)
42

Redefine a-virtual-class with :optimize-slot-access nil:

CL-USER 6 > (defclass a-virtual-class ()
 ((real-slot :initarg :real-slot
   :accessor real-slot
   :initform -1)
 (virtual-slot :accessor virtual-slot
   :initarg :virtual-slot
   :allocation :virtual
   :function
   'a-virtual-class-virtual-slot-function))
 (:metaclass virtual-metaclass)
 (:optimize-slot-access nil))
Warning: (DEFCLASS A-VIRTUAL-CLASS) being redefined in LISTENER
(previously in H:\tmp\vs.lisp).
Warning: (METHOD REAL-SLOT (A-VIRTUAL-CLASS)) being redefined in
LISTENER (previously in H:\tmp\vs.lisp).
Warning: (METHOD (SETF REAL-SLOT) (T A-VIRTUAL-CLASS)) being
redefined in LISTENER (previously in H:\tmp\vs.lisp).
Warning: (METHOD VIRTUAL-SLOT (A-VIRTUAL-CLASS)) being redefined
in LISTENER (previously in H:\tmp\vs.lisp).
Warning: (METHOD (SETF VIRTUAL-SLOT) (T A-VIRTUAL-CLASS)) being
redefined in LISTENER (previously in H:\tmp\vs.lisp).
#<VIRTUAL-METACLASS A-VIRTUAL-CLASS 21AD908C>
Now the standard accessors call `slot-value-using-class`, so we see output when calling `(setf real-slot)

```
CL-USER 7 > (setf (real-slot object) 42)

  setf slot : REAL-SLOT
  42
```
LispWorks supports threads for running computations in parallel. The programming environment, for example, makes extensive use of this mechanism to create separate threads for the various tools.

On Microsoft Windows, Mac OS X, Linux, x86/x64 Solaris, FreeBSD and AIX, LispWorks multiprocessing uses native threads and supports Symmetric Multiprocessing (SMP). The implementation is referred to as "SMP LispWorks" where relevant.

On other platforms LispWorks uses a single native thread and implements user level threads. The implementation is referred to as "non-SMP LispWorks" where relevant.

In SMP LispWorks, Lisp processes (as reported by the Lisp function `ps`) are Operating System threads. These do not necessarily correspond to what system tools show you, for example in Microsoft Windows the Activity monitor shows OS processes, including exactly one for each running LispWorks image.

### 19.1 Introduction to processes

A process (sometimes called a thread) is a separate execution context. It has its own call stack and its own dynamic environment.
A process can be in one of three different states: running, waiting, and inactive. When a process is waiting, it is still active, but is waiting for the system to wake it up and allow its computation to restart. A process that is inactive has stopped, because it has an arrest “reason”.

For a process to be active (that is, running or waiting), it must have at least one run reason and no arrest reasons. If, for example, it was necessary to temporarily stop a process, it could temporarily be given an arrest reason. However the arrest reason mechanism is not commonly used in this manner.

The process that is currently executing is termed “the current process”. The function get-current-process gets the current process, and is the preferred way of doing so. The variable *current-process* is normally bound to the same process, except inside a wait function when it is called by the scheduler.

The current process continues to be executed until either it becomes a waiting process by calling a Process Wait function as described in “Process Waiting and communication between processes” on page 280, or it allows itself to be interrupted by calling process-allow-scheduling (or its current timeslice expires and it involuntarily relinquishes control).

In SMP LispWorks all processes that are not waiting are running as far as LispWorks is concerned, and are scheduled by the operating system to the available CPUs.

In non-SMP LispWorks, the system runs the waiting process with the highest priority. If processes have the same priority then the system treats them equally and fairly. This is called round robin scheduling.

The simplest way to create a process is to use process-run-function. This creates a process with the specified name which commences by applying the specified function to arguments. process-run-function returns immediately and the newly created process runs concurrently.

19.2 Processes basics

19.2.1 Creating a process

To create a new process, use process-run-function.
A process can exit either by returning from the process function or by calling `current-process-kill`.

### 19.2.2 Finding out about processes

The system initializes a number of processes on startup. These processes are specified by `*initial-processes*`.

The current process is obtained by `get-current-process`. A list of all the current processes is returned by `list-all-processes` and the number of them is returned by `processes-count`. The function `ps` is analogous to the UNIX command `ps`, and returns a list of the processes in the system, ordered by priority.

To find a process when you know its name, use `get-process`. To find the name, when you have the process, use `process-name`. The variable `*process-initial-bindings*` specifies the variables that are initially bound in a process.

### 19.2.3 Multiprocessing

To start multiprocessing, use `initialize-multiprocessing`. This function does not return until multiprocessing has terminated.

It is not necessary to use `initialize-multiprocessing` when the LispWorks IDE is already running. Note that, on Windows, Mac OS X, Linux, x86/x64 Solaris, FreeBSD and AIX, the LispWorks images shipped do start the IDE. If you create an image which does not start the IDE, by using the :environment `nil` argument to `save-image`, then multiprocessing can be started in this new image as described below.

#### 19.2.3.1 Starting multiprocessing interactively

You can call `initialize-multiprocessing` from the REPL interface, which generates a default Listener process if no other processes are specified by `*initial-processes*`. 
19.2.3.2 Multiprocessing on startup

There are three ways to make a LispWorks executable start multiprocessing on startup.

1. Use the -multiprocessing command line argument
2. Save an image which starts multiprocessing by doing

   (save-image "mp-lispworks"
   :restart-function 'mp:initialize-multiprocessing)

3. Use delivery to create the executable and pass the argument :multiprocessing t to deliver. The delivery function will be called automatically in a new process. See the LispWorks Delivery User Guide for more details.

LispWorks dynamic libraries always start multiprocessing on startup. See “Multiprocessing in a dynamic library” on page 195 for more information.

In all cases, *initial-processes* can be used to control which processes are created on startup, as described in “Running your own processes on startup” on page 264.

Note: On Windows, Mac OS X, Linux, x86/x64 Solaris, FreeBSD and AIX you cannot save a LispWorks image with multiprocessing running.

19.2.3.3 Running your own processes on startup

*initial-processes* is a list of lists. Each list is used by the system as a set of arguments to process-run-function. During initializing multiprocessing, the system does this:

   (dolist (x mp:*initial-processes*)
     (apply 'mp:process-run-function x))

This script saves a LispWorks image which starts multiprocessing on restart and runs a user-defined process.
19.3 Atomicity and thread-safety of the LispWorks implementation

Access to all Common Lisp objects is thread-safe in the sense that it does not cause an error because of threading issues.

19.3.1 Immutable objects

Immutable (or read-only) objects such as numbers, characters, functions, pathnames and restarts can be freely shared between threads, but special precautions must be taken when all of the following conditions are true:

- A new object is made accessible to other threads ("globally accessible") by storing it in an object that is globally accessible.
- The store that makes it globally accessible is not by \texttt{(setf gethash)}, \texttt{vector-push} or \texttt{vector-push-extend} into a "multithreaded" hash-table or vector (see “Single-thread context arrays and hash-tables” on page 276).
- Other threads read from the globally accessible object without synchronizing with the thread that created it. Synchronizing is typically done by \texttt{lock}, but can also be done by using \texttt{barrier}, \texttt{condition-variable} or \texttt{semaphore}, and by using \texttt{hash-table} locks.

In this situation, it is your responsibility to ensure that all of the stores that occurred when creating the new object are visible to the other threads, as described by “Making an object’s contents accessible to other threads” on page 268.
19.3.2 Mutable objects supporting atomic access

This section outlines for which types of mutable Common Lisp object access is atomic. That is, each value read from the object will correspond to the state at some point in time. Note however, that if several values are read, there is no guarantee about how these values will relate to each other if they are being modified by another thread (see “Issues with order of memory accesses” on page 276).

When one of these mutable atomic objects is modified, readers see either the old or new value (not something else), and it is guaranteed that the Lisp image is not corrupted by the modification even if multiple threads read or write the object simultaneously.

Access to conses, simple arrays except arrays with element type of integer with less than 8 bits, symbols, packages and structures is atomic. Note that this does not apply to non-simple arrays.

Slot access in objects of type standard-object is atomic with respect to modification of the slots and with respect to class redefinition.

vector-pop, vector-push, vector-push-extend, (setf fill-pointer) and adjust-array are all atomic with respect to each other, and with respect to other access to the array elements.

The Common Lisp functions that access hash tables are atomic with respect to each other. See also modify-hash for atomic reading and writing an entry and with-hash-table-locked. See also “Modifying a hash table with multiprocessing” on page 279 for thread-safe ways to ensure a table entry.

Access to packages is atomic.

Note that pathnames cannot be modified, and therefore access to them is always atomic.

Access to synchronization objects (of type mailbox, barrier, semaphore and condition-variable) is atomic. More information about these objects is in “Synchronization between threads” on page 285.

Operations on editor buffers (including points) are atomic and thread-safe as long as their arguments are valid. This includes modification to the text. However, buffers and points may become invalid because of execution on another thread. The macros editor:with-buffer-locked and editor:with-point-
locked should be used around editor operations on buffers and points that may be affected by other processes. Note that this is applicable also to operations that do not actually modify the text, because they can behave inconsistently if the buffer they are looking at changes during the operation. See the LispWorks Editor User Guide for details of these macros.

19.3.3 Mutable objects not supporting atomic access

This section outlines for which types of mutable Common Lisp object access is not atomic.

Access to arrays with element type of integer of less than 8 bits is not guaranteed to be atomic.

Access to non-simple arrays is not guaranteed to be atomic.

Access to lists (including alists and plists) is not atomic. Lists are made of multiple cons objects, so although access to the individual conses is atomic, the same does not hold for the list as a whole.

Sequence operations which modify multiple elements are not atomic.

Macros that expand to multiple accesses are in general not atomic. In particular, modifying macros like push and incf are not atomic (but see the atomic versions of some of them in “Low level atomic operations” on page 299).

Making several calls to Common Lisp functions that access hash tables will not be atomic overall. However LispWorks provides thread-safe ways to ensure a hash table entry - see “Modifying a hash table with multiprocessing” on page 279. See also modify-hash for atomic reading and writing an entry and with-hash-table-locked.

Stream operations are in general not atomic. There is an undocumented interface for locking of streams when this is required - contact Lisp Support if you need this.

Operations on CAPI objects are not atomic in general. The same is true for anything in the LispWorks IDE. These operations need to be invoked from the thread that owns the object, for example by capi:execute-with-interface or capi:apply-in-pane-process.
19.3.4 Making an object's contents accessible to other threads

An object's contents become accessible to other threads when it is stored into a cell that may be accessed by those threads (a "globally accessible cell"). In most cases, you should either use a synchronization mechanism (typically a lock) to control access to the cell because it is the most reliable approach or use a mailbox to make the object accessible to other threads (the mailbox acts as the globally accessible cell).

If you do not use a synchronization mechanism or mailbox, then all stores into the object must be forced to be visible to other threads ("ensured") before the object is stored in the globally accessible cell. This also applies to any stores that LispWorks did during construction of the object. Normally, LispWorks does not do that for every store, because it would slow the program too much. Note that numbers, except fixnum and short-float (in 32-bit LispWorks) or fixnum and single-float (in 64-bit LispWorks), are also objects that are constructed using stores that need to be ensured.

Storing objects into globally accessible cells would typically be done by setf or a related macro such as push or incf, but can also be done by Common Lisp functions such as rplaca, fill, nsublis, nsubst or replace (when the target is globally accessible). If stores into the objects that these functions store have not been ensured and may be read by another thread without synchronization, then one of the mechanisms in “Ways to guarantee the visibility of stores” on page 268 must be used. Note that when a sequence itself, rather than the elements, is modified, for example by delete, nreverse, nconc, nunion, then all access to the sequence needs to controlled by a synchronization mechanisms, which will also guarantee the visibility of stores.

19.3.4.1 Ways to guarantee the visibility of stores

The visibility (in other threads) of stores in an object referenced by a globally accessible cell can only be guaranteed in these situations:

1. You use a lock or any other synchronization mechanism (barrier, condition-variable, semaphore, mailbox, or the lock of a hash-table) to serialize all access to the globally accessible cell. Any use of a synchronization mechanism that may affect the behavior of another thread will implicitly ensure all preceding stores on the current thread.
2. The store is done by \texttt{(setf symbol-function)} or \texttt{(setf macro-function)} into a symbol, or by one of \texttt{(setf gethash)}, \texttt{vector-push}, \texttt{vector-push-extend} into a "multithreaded" hash-table or vector (see “Single-thread context arrays and hash-tables” on page 276).

3. You store a newly interned symbol (all stores that occur during interning are ensured). However, if the store is done using an operator that allocates (see “Special care for macros and accessors that may themselves allocate” on page 270) then you will still need to ensure.

4. You use one of the low level atomic operations, all of which ensure all stores on the current thread before they modify the cell. This includes stores from any allocation they may do and applies also to user defined atomic modify macros that are defined by \texttt{define-atomic-modify-macro}. Currently these atomic operations are: \texttt{compare-and-swap}, \texttt{atomic-exchange}, \texttt{atomic-push}, \texttt{atomic-pop}, \texttt{atomic-fixnum-incf}, \texttt{atomic-fixnum-decf}, \texttt{atomic-incf}, \texttt{atomic-decf}. Any atomic operations added in the future will do the same.

5. You start a new thread and access the object from that thread. LispWorks ensures all preceding stores on the current thread before the new thread runs, so if all accesses to an object occur in threads that start after the object was last modified then you do not need to ensure the stores into it.

6. You are storing an immediate object (\texttt{fixnum}, \texttt{character} or \texttt{short-float} in 32-bit LispWorks; \texttt{fixnum}, \texttt{character} or \texttt{single-float} in 64-bit LispWorks) in the globally accessible cell. There are no stores that need to be ensured during the creation of these objects. However, if the store is done using an operator that allocates (see “Special care for macros and accessors that may themselves allocate” on page 270) then you will still need to ensure.

7. The store is by \texttt{setf} or a related macro (for example \texttt{push} or \texttt{incf}), and the \texttt{place} argument is wrapped by the macro \texttt{globally-accessible}.

8. You call \texttt{ensure-stores-after-stores} between the time the object was made (and any stores of interest were done into it) and the time it is stored into a globally accessible cell. \texttt{ensure-stores-after-stores} ensures all preceding stores in the current thread.
Note: ensure-memory-after-store and ensure-stores-after-memory do what ensure-stores-after-stores does and more, but may be more expensive and are not required in this context.

Stores into objects must be "ensured" once by one of the above mechanisms, before the object becomes globally accessible. Stores that occur after this are not guaranteed to be visible to other threads until another ensuring operation.

In some circumstances you can make the program more efficient by explicitly ensuring stores using globally-accessible (7) or ensure-stores-after-memory (8) before or when the object is first made visible. See “Ensuring stores are visible to other threads” on page 271 for more details.

A synchronizing operation (1), atomic operation (4) or a call to ensure-stores-after-stores (8) ensures all stores into objects that were created by the current thread. The other situations ensure the stores that they perform and anything pointed to by those stores, but are not guaranteed to ensure other stores (because they may be able to skip ensuring in some circumstances).

19.3.4.2 Special care for macros and accessors that may themselves allocate

A situation that always requires special care is storing into a globally accessible cell using macros and accessors that may themselves allocate. If the store is not done using (1), (4) or (5) in “Ways to guarantee the visibility of stores” on page 268, then you need to use globally-accessible to ensure the stores in new objects that may be allocated are visible, even if the object that is stored does not need it (because it is an immediate, an interned symbol or was ensured earlier). These macros and accessors include:

Macros: push, pushnew, push-end, push-end-new, incf (when not a fixnum), decf (when not a fixnum).

Accessors: getf, cdr-assoc, mask-field (when not a fixnum), ldb (when not a fixnum).
19.3 Atomicity and thread-safety of the LispWorks implementation

User defined accessors that allocate:

Any user defined accessor (that is an operator with a setf expander defined by define-setf-expander or defsetf) that allocates during the setting operation. Note that allocation during the macro expansion is not an issue.

See “Destructive macros and accessors that allocate internally” on page 274 for more details.

19.3.5 Ensuring stores are visible to other threads

A store to a cell from one thread is said to be “visible” from another thread when a load from that cell from the other thread obtains the value that was stored. Within a single thread, all stores are visible immediately to loads from the same thread, but that is not always the case in a multithreaded situation. Store operations that occur in one thread are not necessarily visible from other threads until something ensures that they are. In other words, another thread loading from the cell where the store was done may still obtain the cell's previous value, even if "logically" it seems that the load happened after the store. For a new object, the previous value may be anything that was in memory, including an invalid value that may cause crashes.

19.3.5.1 An example to consider the issues

For example, assume that the symbol *a-global-symbol* is not dynamically bound anywhere and its value is nil, and we have two threads, A and B, executing without synchronization.

Thread A executes this code:

```lisp
(setq *a-global-symbol* (cons 1 2))
```

and thread B executes this code:

```lisp
(let ((maybe-cons *a-global-symbol*))
  (if (consp maybe-cons)
      (car maybe-cons)
      -1))
```
It looks like the form that thread B executes will always return either 1 (if it happens after thread A has set \texttt{*a-global-symbol*}) or -1 (otherwise), because in the case that \texttt{maybe-cons} is a cons it must already have 1 in the car. However, that is not necessarily true because the store of the cons into \texttt{*a-global-symbol*} may be visible to thread B before the store of 1 into the cons (which happens inside the call to \texttt{cons}) is visible. This applies to explicit stores in the program as well, for example if thread A executes:

\begin{verbatim}
(setq *a-global-symbol* (rplaca (list nil) 1))
\end{verbatim}

then the same problem arises (in this case, the call to \texttt{car} in thread B may return 1, \texttt{nil} (the value that \texttt{list} stored in the car), or whatever was in that memory before that).

Note: the second load in thread B (inside \texttt{car}) is dependent on the first load (reading the value cell from \texttt{*a-global-symbol*}). Such dependent loads are guaranteed to occur in the program order in all current LispWorks releases. In situations when the two loads are independent, but you still need them to occur in the program order, you will need to use \texttt{ensure-loads-after-loads}.

### 19.3.5.2 The general solution using a lock

In most circumstances, all access to globally accessible cells should be controlled by a \texttt{lock}, which eliminates all of these problems because releasing a lock implicitly ensures that all stores in that thread are visible to all other threads, so by the time another thread gets ownership of the lock, all the stores are already visible. Sending an object via a \texttt{mailbox}, using \texttt{(setf gethash)}, \texttt{vector-push} or \texttt{vector-push-extend} or synchronizing using any of the other synchronization mechanisms (\texttt{barrier}, \texttt{condition-variable}, \texttt{semaphore} or the lock of a \texttt{hash-table}) also ensures the stores are already visible (for a full list, see “Making an object’s contents accessible to other threads” on page 268). If you make an object globally accessible without any of these mechanisms, then you need to ensure explicitly that the stores are all visible. Note that if the store occurs inside a lock but the object is not read inside the same lock, then you still need to ensure the stores, because the reading may happen before unlocking has ensured the stores are visible. Note also that, for some macros and accessors (listed below), you need to ensure the stores even if the value that you store does not need ensuring.
19.3.5.3 An alternative solution using globally-accessible

If you need to explicitly ensure that all stores are visible, then the best approach is to use globally-accessible, which takes a single argument, place, which can be any generalized reference form as described in section 5.1.1 Overview of Places and Generalized Reference of the Common Lisp HyperSpec. In most cases, (globally-accessible place) is the same as place. However, when globally-accessible is used inside setf or a related macro such as push or incf then it also ensures all stores are visible to other threads before modifying place. For example, if we change the code that thread A in Section 19.3.5.1 executes to:

```
(setf (sys:globally-accessible *a-global-symbol*)
     (cons 1 2))
```

then the value returned by the form in thread B is guaranteed to be 1 or -1 as expected.

19.3.5.4 An alternative solution using ensure-stores-after-stores

The other approach is to use ensure-stores-after-stores just before storing into the globally accessible cell, but after any allocation or other operations that modify the object being stored in the cell. In the example in Section 19.3.5.1, thread A would do:

```
(let ((cons (cons 1 2)))
  (sys:ensure-stores-after-stores)
  (setq *a-global-symbol* cons))
```

ensure-stores-after-stores cannot be used when the form that stores the object is a destructive macro such as push (for a full list, see “Destructive macros and accessors that allocate internally” on page 274 below), because push itself allocates a cons using stores that need to be ensured. globally-accessible takes care of that, by ensure the stores after any allocation that the macro may do (except when it is a macro with a non-standard setf expansion that allocates or stores in the setter). Thus globally-accessible is preferable when you can use it. You need to use ensure-stores-after-stores when the store is encapsulated in some other operation. For example, if you use fill to store a new object in a globally accessible sequence (some-vector below) then you will need to do something like this:
(let ((new-object (cons x y)))
    (sys:ensure-stores-after-stores)
    (fill some-vector new-object))

**ensure-stores-after-stores** always ensures all the stores that have happened in the current thread. **globally-accessible** is not guaranteed to ensure all preceding stores accept into the value that it stores, because it may be able to skip ensuring in some circumstances.

### 19.3.5.5 Destructive macros and accessors that allocate internally

The macros **push**, **pushnew**, **push-end**, **push-end-new**, **incf** (when not a fixnum) and **decf** (when not a fixnum) may generate new objects internally, so if they are used to destructively modify a globally accessible cell without synchronization then you will need to use **globally-accessible**.

For example:

```lisp
(pushnew some-object (sys:globally-accessible *a-global-symbol*))
```

Note that **globally-accessible** is needed with **push**, **pushnew**, **push-end**, and **push-end-new** even if there are no stores into the object being pushed that need ensuring. For **incf** and **decf**, if you can guarantee that the new value is fixnum, then you do not need **globally-accessible**.

The accessors **getf**, **cdr-assoc**, **mask-field** and **ldb** take a **place** argument that may generate new objects when modified, so if **place** is globally accessible and it is modified without synchronization then you will need to wrap **globally-accessible** around such modifications of **place**.

For example:

```lisp
(setf (getf (sys:globally-accessible *a-global-symbol*)
            :key)
      value)
```

For **getf** and **cdr-assoc**, **globally-accessible** is needed even if there are no stores into the new object and key that need ensuring because new conses might be added to the **place**. For **mask-field** and **ldb**, if you are absolutely sure that the new value is fixnum then you do not need **globally-accessible**.
In addition, `setf` expanders defined by `define-setf-expander` or `defsetf` cannot be used on globally accessible cells without synchronization (by a lock or other synchronization mechanism) if they do any of the following:

- allocate inside the value forms (the second value returned by `get-setf-expansion`);
- allocate inside the storing form (the fourth value returned by `get-setf-expansion`);
- store to anywhere except the `place`.

See section 5.1.1.2 Setf Expansions of the Common Lisp HyperSpec for the definition of "value forms" and "storing form".

### 19.3.5.6 Miscellaneous notes

For an object that is not modified later, ensuring all stores when the object is created is sufficient, and after that the object can be used freely from any thread.

`ensure-stores-after-stores` can be used to ensure stores for objects that are modified after becoming globally accessible. However, if you need to ensure that the new values are seen by other threads that may be already accessing the modified objects then you need to use some synchronization mechanism anyway. Thus in most cases you should use a lock, which will deal with the synchronization.

`ensure-stores-after-stores` does slow the program a little on architectures that need it (Currently ARM, ARM64 and PowerPC), so you can consider the following optimizations:

- If you construct several objects that all need to be stored into globally accessible cells, then you can reduce the overhead by making them all, calling `ensure-stores-after-stores` once and then storing them all.
- If the store is done while holding a lock but the load is done without the lock, and loading happens less frequently than storing, then you may consider loading while holding the lock as well to avoid needing to explicitly ensure the stores.
As noted in “Ensuring stores are visible to other threads” on page 271, loads from an object that was obtained from a globally accessible cell are currently guaranteed to occur after the load the cell itself, because all the architectures that LispWorks runs on guarantee that. In principle, sometime in the future there may be a new architecture that does not provide that guarantee. You can guard against this by using `globally-accessible` when reading from the globally accessible cell as well. Currently that just macroexpands into its argument, so does not affect the performance, but for an architecture that requires anything it will do the right thing.

### 19.3.6 Issues with order of memory accesses

When multiple threads access the same memory location, the order of those accesses is not generally guaranteed. You should therefore not attempt to implement "lockless algorithms" which depend on the order of memory accesses unless you have a good understanding of multiprocessing issues at the CPU level (see “Ensuring order of memory between operations in different threads” on page 301).

However, all of the “Low level atomic operations” and locking operations (see “Locks”) do ensure that all memory accesses that happen before them have finished and that all memory accesses that happen after them start after them. Therefore, normally there is nothing special to consider when using those operations. The modification check macros described in “Aids for implementing modification checks” also take care of this.

### 19.3.7 Single-thread context arrays and hash-tables

Access to hash tables and non-simple arrays can be improved where they are known to be accessed in a single thread context. That is, only one thread at the same time accesses them.

The `make-hash-table` argument `single-thread` tells `make-hash-table` that the table is going to be used only in single thread context, and therefore does not need to be thread-safe. Such a table allows faster access.

Similarly the `make-array` argument `single-thread` creates an array that is single threaded. Currently, the main effect of `single-thread` is on the speed of `vector-pop`, `vector-push`, and `vector-push-extend` on non-simple vectors. These
operations are much faster on "single threaded" vectors, typically more than twice as fast as "multithreaded" vectors.

You can also make an array be "single-threaded" with \texttt{set-array-single-thread-p}.

The result of parallel access to a "single-threaded" vector is unpredictable.

\section*{19.4 Locks}

Locks can be used to control access to shared data by several processes.

The two main operators used in locking are the function \texttt{make-lock}, to create a lock, and the macro \texttt{with-lock}, to execute a body of code while holding the specified lock.

A lock has a name (a string) and several other components. The printed representation of a lock shows the name of the lock and whether it is currently locked. Additionally if the lock is locked it shows the name of the process holding the lock, and how many times that process has locked it. For example:

\begin{verbatim}
&#<MP:LOCK "my-lock" Locked 2 times by "My Process" 2008CAD8>
\end{verbatim}

The function \texttt{lock-owner} returns the process that locked a given lock.

The function \texttt{lock-name} returns the name of a lock.

The function \texttt{process-lock} blocks the current process until a given lock is claimed or a timeout passes, and \texttt{process-unlock} releases the lock.

The macro \texttt{with-lock} executes code with a lock held, and releases the lock on exit, as if by \texttt{process-lock} and \texttt{process-unlock}.

If you need to avoid blocking on a lock that is held by some other thread, then use \texttt{with-lock} with \texttt{timeout 0}, like this:

\begin{verbatim}
(unless (mp:with-lock (lock nil 0) (code-to-run-if-locked) t) (code-to-run-if-not-locked))
\end{verbatim}

The macros \texttt{with-sharing-lock} and \texttt{with-exclusive-lock} can be used with sharing locks.
19.4.1 Recursive and sharing locks

The keyword argument recursivep to make-lock, when true, allows the lock to be locked recursively. recursivep is true by default. If recursivep is false then trying to lock again causes an error. This is useful for debugging code where the lock is not expected to be claimed recursively.

The keyword argument sharing to make-lock, when true, creates an "sharing" lock object, which supports sharing and exclusive locking. A sharing lock is handled by different functions and methods. See with-exclusive-lock, with-sharing-lock, process-exclusive-lock, process-exclusive-unlock, process-sharing-lock and process-sharing-unlock.

19.4.2 Querying locks


19.4.3 Guarantees and limitations when locking and unlocking

In compiled code process-lock, process-exclusive-lock and process-sharing-lock are guaranteed to return if they locked their argument. In other words there will not be any throw between the time they locked the lock and the time they return. That means that in compiled code the next form will at least start executing, and if it is an unwind-protect the cleanup forms will at least start executing. (If the code is evaluated, this is not guaranteed.) "Locking" here also means incrementing the count of a lock that is already held by the current thread.

However these functions may throw before locking. For example, in the following code process-lock may throw without locking, for example because something interrupts the process by process-interrupt:

```lisp
(unwind-protect
  (progn (mp:process-lock lock)
         (whatever))
  (mp:process-unlock lock))
```

If this call to process-lock does throw without locking, then process-unlock will be called on a lock that is not locked.
The correct code that guarantees (when compiled) that `process-unlock` is called on exit only when `process-lock` did lock is:

\[
\begin{align*}
\text{(mp:process-lock lock)} \\
\text{(unwind-protect (whatever)) (mp:process-unlock lock)}
\end{align*}
\]

Conversely, `process-unlock`, `process-exclusive-unlock` and `process-sharing-unlock` guarantee to successfully unlock the lock, but are not guaranteed to return.

For example, the following code may fail to call `another-cleanup`:

\[
\begin{align*}
\text{(mp:process-lock lock)} \\
\text{(unwind-protect (whatever)) (mp:process-unlock lock) (another-cleanup)}
\end{align*}
\]

If `another-cleanup` is essential to execute in all throws, it needs its own `unwind-protect`:

\[
\begin{align*}
\text{(mp:process-lock lock)} \\
\text{(unwind-protect (whatever) (unwind-protect (mp:process-unlock lock) (another-cleanup)))}
\end{align*}
\]

**Note:** the guarantees described in this section are relevant only in compiled code.

### 19.5 Modifying a hash table with multiprocessing

Each `hash-table` access operation is thread-safe and atomic, but there is no guarantee of atomicity between access operations.

The modify macros (for example `incf`) all expand to two access operations, reading the value and writing the modified value, and are therefore not atomic. They need to be either done while holding a `lock`, or using `modify-hash`.
Another common operation is "ensuring an entry", that is reading and, if reading fails, adding a value to the table. For example:

```lisp
(or (gethash key hash-table)
    (setf (gethash key hash-table) (construct-new-value)))
```

If two threads do that in parallel, one of them may end up with a value that is not in the table. One solution is to lock that table:

```lisp
(with-hash-table-locked hash-table
  (or (gethash key hash-table)
      (setf (gethash key hash-table) (construct-new-value))))
```

However that always locks the table, which is inefficient. The correct way to do it is either to do:

```lisp
(or (gethash key hash-table) ; first try without the lock
    (with-hash-table-locked hash-table
      (or (gethash key hash-table) ; check again inside the lock
          (setf (gethash key hash-table) (construct-new-value)))))
```

or use `gethash-ensuring` or `with-ensuring-gethash`.

## 19.6 Process Waiting and communication between processes

Process Waiting means that a process suspends its own execution until some condition is true. The generic Process Wait functions take a `wait-function` argument, which is arbitrary though somewhat restricted Lisp code. A process resumes running when the `wait-function` returns true. The specific Process Wait functions wait for a specific condition.

### 19.6.1 Specific Process Wait functions

For communication between processes, these are:

- `mailbox-read`, `process-wait-for-event`, `mailbox-wait` and `mailbox-wait-for-event`.

For synchronization, these are:

- `condition-variable-wait` and `barrier-wait`, also `semaphore-acquire` and `semaphore-release`. 
For locking these are:

process-lock, process-exclusive-lock and process-sharing-lock.

For sleeping, these are:

c1:sleep and current-process-pause.

The specific Process Wait functions are designed to reduce latencies and to increase efficiency. In particular, in SMP LispWorks they should be used in preference to the generic Process Wait functions.

19.6.2 Generic Process Wait functions

The generic Process Wait functions are:

process-wait and process-wait-with-timeout

process-wait-local and process-wait-local-with-timeout


Note: For brevity we sometimes refer to "the *-periodic-checks functions" or "the *-with-timeout functions".

All the generic Process Wait functions take wait-reason and wait-function arguments and potentially also arguments to pass to the wait-function. The *-with-timeout functions mentioned above also take a timeout argument. The *-periodic-checks functions also take a period argument.

The wait-reason is used only to mark the process as waiting for something for debugging purposes. It does not affect the behavior of the functions.

The generic Process Wait functions "wake up" (that is, they simply return to the caller) either when the timeout passed (if they take a timeout argument), or when the wait function returns true. The three pairs of functions mentioned above differ in the mechanism that calls the wait function.

process-wait and process-wait-with-timeout arrange that the "scheduler" will call the wait function when it runs. The "scheduler" is invoked at various points, in an indeterminate process. The advantage of this is that the programmer does not need to worry too much about when the wait function is going to be called. In non-SMP LispWorks (that is, LispWorks 5.1 and ear-
lier) the programmer does not need to worry at all: when some process sets up something that would make the wait function return true, the waiter process could not run anyway until the setting-up process stopped for some reason (including preemption), by which time the scheduler would have called the wait function if it had not done it before. In SMP LispWorks (that is, LispWorks 6.0 and later), these two processes can run simultaneously, so the delay between the setting up and the scheduling is not necessary. It can be avoided by "poking" the waiting process with `process-poke`, if the waiting process is known, or by invoking the scheduler by `process-allow-scheduling`.

**Note:** All the specific Process Wait functions, described in “Specific Process Wait functions” on page 280, record that they wait, and the operations that allow them to continue implicitly "poke" the waiting process. Therefore the specific functions avoid the problem of latency or needing `process-poke`, and should be used in preference where possible.

A large disadvantage of `process-wait` and `process-wait-with-timeout` is that their `wait-function` is called by the "scheduler" in an indeterminate process. That means that the wait function does not see the dynamic environment of the calling process (including error handlers), and cannot be debugged properly. It is also called often, and so it needs to be reasonably fast and not allocate much. In addition, having to call the wait function adds overhead to the system. Therefore in general, if you can achieve the required effect by using either any of the specific wait functions or a `process-wait-local*` function, you should do that and avoid `process-wait` and `process-wait-with-timeout`.

`process-wait-local` and `process-wait-local-with-timeout` do not have all the disadvantages listed above, but their `wait-function` is called only when the process is poked (or at the end of the `timeout`). That means that the programmer does need to worry about when they are called. Typically some other process will set up something, and then poke the waiting process to check that it can run.

**Note:** if the setting up process always knows for sure whether the waiting process can run, then it is normally simpler to use one of the specific Process Wait functions, or maybe even `process-stop` and `process-unstop`.

The `*-periodic-checks` functions give a partial solution to the question of calling the wait function, by ensuring there is a maximum period of time between calls. If having a bounded delay where a bound of more than 0.1 sec-
ond is not a problem, then the **-periodic-checks** functions are a simple and efficient way to achieve it.

When the delays need to be bounded by a shorter period, either one of the specific Process Wait functions or explicit calls to `process-poke` need to be used. The latter combined with `process-wait-local` is the most efficient mechanism, but it does require the programmer to ensure that `process-poke` is called in all the right places.

### 19.6.3 Communication between processes and synchronization

The simplest way to pass a specific event between two processes it to use `process-wait-for-event` on the receiving process, and `process-send` on the sender side. The "event" that is passed is can be any Lisp object.

`process-send` and `process-wait-for-event` use a mailbox to pass the object (the `process-mailbox` of the receiver). It is possible to use a mailbox object directly, and to communicate between multiple senders and receivers. Use `make-mailbox` to make a mailbox, and `mailbox-send` to put a message in it. In some situations there may be an imbalance between sending and receiving messages in a mailbox, which may cause the mailbox to become very big. When this is a problem, you can use `mailbox-send-limited` to make the sending process wait (or do something else) once the mailbox grows to some limit. There are also functions `mailbox-count`, `mailbox-size` and `mailbox-full-p` to help with these situations.

The receiver(s) use `mailbox-wait-for-event`, `mailbox-wait` or `mailbox-read`. `mailbox-wait-for-event` should be used on processes that may make windows (including any process associated with a CAPI interface), but can be used elsewhere. `mailbox-read` is faster, but if it used on a process with a window it may cause hanging.

In general, the receiving process decides hows to interpret an event. However, the system has a "standard" generic function, `general-handle-event`, to interpret events. `general-handle-event` has methods that process lists by applying the `car` to the `cdr`, and processes function objects or symbols by calling them. There is a method on `t` that does nothing. You can add your own method on your defined classes (which can be structures).
general-handle-event is used when system code needs to interpret events, most importantly processes that CAPI uses to display windows use it. Hence for processes that use the "standard" event handling, you can send an object using process-send and expect it to be processed by general-handle-event. general-handle-event is also used by process-all-events, which processes all the events for the current process, and wait-processing-events, which waits until some predicate returns true while processing events.

In some situations it is useful to execute some code next time the current process processes events, rather than immediately. That can be achieved by process-send with the current process, or more conveniently by current-process-send.

process-wait-for-event and process-send and mailbox are the primary interface for communication between processes, and should be used unless there is a very good reason to use a different mechanism.

19.6.4 Synchronization

Synchronization can be achieved by the various process-wait* functions with the appropriate wait-function argument, but for simple cases of synchronization it is better to use the synchronization objects: condition-variable or barrier. These synchronization objects are simple, efficient, deal with all thread-safety issues, and ensure that the processes that are ready to run will run immediately, rather than the next time that the wait function is called.

Condition variables (or type condition-variable) are used when one or more processes have the knowledge to control when another process(es) runs. The "ignorant" process(es) use condition-variable-wait to wait until they can continue. The "knowledgeable" process(es) use condition-variable-signal and condition-variable-broadcast to tell the "ignorant" processes when they can run. Because the communication is via the condition variable, the processes do not need to know explicitly about each other. For more details, see “Condition variables” on page 285.

Barriers (of type barrier) are used (mainly) for symmetric synchronization, when a group of processes needs to ensure that none of them goes too far ahead of the rest. The processes call barrier-wait when they want to synchronize, and barrier-wait waits until the other process arrive too (that is,
they call \texttt{barrier-wait}). Barriers have additional features that allow more complex synchronization. For more details, see “Synchronization barriers” on page 287.

19.7 \textbf{Synchronization between threads}

In LispWorks 5.1 and previous versions, the main way to synchronize between threads is to use \texttt{mp:process-wait} or \texttt{mp:process-wait-with-timeout} to supply a predicate to the scheduler. The predicate runs periodically in the background to identify threads that are no longer blocked.

These functions are still available, but there are some alternatives that can be more efficient in many cases by removing the need for the scheduler. The alternatives are:

- Mailboxes (FIFO queues). See \texttt{make-mailbox} and \texttt{mailbox-send}.
- Condition Variables (used with a \texttt{lock}). See “Condition variables” on page 285.
- Barriers (counting arrivals at a certain point in the code). See “Synchronization barriers” on page 287.
- Counting Semaphores (limiting the number of users of a shared resource). See “Counting semaphores” on page 287.

Access to all of these objects is atomic and does not require additional locks (except for the \texttt{lock} that is already used with a \texttt{condition-variable}).

19.7.1 \textbf{Condition variables}

A \texttt{condition-variable} allows you to wait for some condition to be satisfied, based on the values stored in shared data that is protected by a \texttt{lock}. The condition is typically something like data becoming available in a queue.

The function \texttt{condition-variable-wait} is used to wait for a \texttt{condition-variable} to be signaled. It is always called with the \texttt{lock} held, which is automatically released while waiting and reclaimed before continuing. More than one thread can wait for a particular \texttt{condition-variable}, so after being notified about the condition changing, you should check the shared data to see if it represents a useful state and call \texttt{condition-variable-wait} again if not.
The function `condition-variable-signal` is used to wake exactly one thread that is waiting for the `condition-variable`. If no threads are waiting, then nothing happens.

Alternatively, the function `condition-variable-broadcast` can be used to wake all of the threads that are waiting at the time it is called.

Any threads that wait after the call to `condition-variable-signal` or `condition-variable-broadcast` will not be woken until the next call.

In most uses of condition variables, the call to `condition-variable-signal` or `condition-variable-broadcast` should be made while holding the lock that waiter used when calling `condition-variable-wait` for this `condition-variable`. This ensures that the signal is not lost if another thread is just about to call `condition-variable-wait`.

The function `condition-variable-wait-count` can be used to determine the current number of threads waiting for a `condition-variable`.

The `condition-variable` implementation in LispWorks aims to comply with the POSIX standard where possible.

`condition-variable-wait`, `condition-variable-signal` and `condition-variable-broadcast` have corresponding functions `lock-and-condition-variable-wait`, `lock-and-condition-variable-signal` and `lock-and-condition-variable-broadcast`. For `condition-variable-wait` there is also `simple-lock-and-condition-variable-wait`, which is simpler to use.

The `lock-and-condition-...` functions perform the equivalent of locking and in the scope of the lock doing something and calling the corresponding `condition-...` function.

The `lock-and-condition-...` functions not only make it simpler to code, they also make it easier to avoid mistakes, and can optimize some cases (in particular, the quite common case when there is no need to lock on exit from `condition-variable-wait`). They are the recommended interface.

The `lock-and-condition-...` functions can be used together with `condition-...` functions on the same locks and `condition-variables`.

Note: In cases when only one process waits for the condition, using `process-wait-local` for waiting and `process-poke` for signaling is easier, and involves less overhead.
19.7 Synchronization between threads

19.7.2 Synchronization barriers

Barriers are objects of type `barrier` that are used to synchronize multiple threads. A `barrier` has a `count` that determines how many "arrivals" (calls to `barrier-wait`) have to occur before these calls return.

The main usage of barriers is to ensure that a group of threads have all finished some stage of an algorithm before any of them proceeds.

The typical way of using a `barrier` is to make one with a `count` that is the same as the number of threads that are going to work in parallel and then create the threads to do the work. When each thread has done its work, it synchronizes with the others by calling `barrier-wait`. In most cases `barrier-wait` is the only barrier API that is used.

For example, assume you have a task that be broken into two stages, where each stage can be done in parallel by several threads, but the first stage must be completely finished before any processing of the second stage can start. Then the code will do:

```lisp
(let ((barrier (mp:make-barrier num-of-processes)))
  (do-times (p num-of-processes)
    (mp:process-run-function (format nil "Task worker -d" p)
      ()
      #'(lambda (process-number barrier)
          (do-first-stage process-number)
          (mp:barrier-wait barrier)
          (do-second-stage process-number))
      p
    barrier)))
```

It is also possible to use a `barrier` to block an indefinite number (up to `most-positive-fixnum`) of processes, until another process decides that they can go. For this the `barrier` is made with `count` `t` (or `most-positive-fixnum`). The other process then uses `barrier-disable` to "open" the `barrier`. If required, the `barrier` can be enabled again by `barrier-enable`.

See also `barrier-block-and-wait`.

19.7.3 Counting semaphores

A counting semaphore is a synchronization object that allows different threads to coordinate their use of a shared resource that contains some num-
ber of available units. The meaning of each unit depends on what the semaphore is being used to synchronize.

The three main functions associated with semaphores are: make-semaphore, which makes a new semaphore object; semaphore-acquire, which acquires units from a semaphore and semaphore-release, which releases units back to a semaphore. The current thread will block if it attempts to acquire more units than are current available.

The functions semaphore-name, semaphore-count and semaphore-wait-count can be used to query the name, available unit count and count of waiting units from a semaphore.

19.8 Killing a process, interrupts and blocking interrupts

19.8.1 Killing a process

When the function of the process (third argument to process-run-function) returns the process exits, but in many cases it is more convenient to terminate the process without returning all the way to the process function.

The function current-process-kill can be used to kill the current process. It executes all the unwind forms on the stack first. Checking in appropriate places and calling current-process-kill is a convenient and safe way (as long as there are unwind-protect forms where needed) of causing processes to exit when they should.

process-terminate can be used to kill any process. If there is no Terminate Method (see current-process-set-terminate-method) it uses the process interrupting mechanism, so if the other process blocks interrupts it will continue to run until it stops blocking. Because the killing interrupt can happen inside unwind forms of unwind-protect (unless they are executed with interrupts blocked) process-terminate is not safe unless all essential unwinding forms are executed with interrupts blocked. In most cases it is probably easier to not use process-terminate in actual applications.
19.8.2 Interrupting a process

`process-interrupt` and `process-interrupt-list` can be used to interrupt a process and execute arbitrary code. Since the interrupt happens at a "random" time, it should have minimal interaction with any data structures that are being modified. For robust applications it is probably better never to use it except during development.

19.8.3 Blocking interrupts

The purpose of blocking interrupts is to prevent a process aborting in the middle of an operation that needs to be completed. A typical example is the cleanup forms of an `unwind-protect`.

Blocking interrupts does **not** provide atomicity. Other processes may continue to execute.

Blocking interrupts limits the control that LispWorks has over the processes, so interrupts should not be blocked except when necessary. However, apart from blocking interrupts in a process it does not affect the behavior of the system.

The following macros and functions allow control over blocking interrupts: `allowing-block-interrupts`, `with-interrupts-blocked`, `current-process-unblock-interrupts` and `current-process-block-interrupts`.

Additionally the macros `unwind-protect-blocking-interrupts` and `unwind-protect-blocking-interrupts-in-cleanups` allow your program to prevent interrupts from stopping cleanup forms from completing.

Compatibility note: In LispWorks 5.1 and previous versions, `mp:without-preemption` and `mp:without-interrupts` are sometimes used to block interrupts, but they also provide atomicity. In many cases (probably most), they are used to provide atomicity, and in these cases they cannot be replaced by blocking interrupts. To get atomicity in LispWorks 6.0 and later you need to use locks or atomic operations. To get atomicity while debugging, you can also use `with-other-threads-disabled`.
19.8.4 Old interrupt blocking APIs removed

The macros `mp:without-interrupts` and `mp:without-preemption`, which were available in LispWorks 5.1 and earlier, are no longer supported. The semantics of these macros allowed them to be used for several different purposes, which now require specific solutions.

- Atomic operations. This use was designed to make operations atomic with respect to other uses of the same macro or with respect to some other unquantified operations that were expected to be atomic, such as reading or writing a single slot in an object. Code of this kind should be converted to use locks (see “Locks” on page 277) or low level atomic operations (see “Low level atomic operations” on page 299).
- Complete operations. This use was designed to ensure that a set of operations completed without being interrupted by `mp:process-interrupt`, keyboard breaks and so on. See “Blocking interrupts” on page 289 for the new approach.

The following subsections show examples of typical uses of the old interrupt blocking APIs together with their replacements. The examples use `mp:without-interrupts` but the ideas also apply to uses of `mp:without-preemption`.

### 19.8.4.1 Atomic increment

**Old:**

```lisp
(without-interrupts
  (incf *global-counter*))
```

**New:** use low level atomic operations.

```lisp
(sys:atomic-incf *global-counter*)
```

### 19.8.4.2 Atomic push/pop

**Old:**

```lisp
(without-interrupts
  (push value *global-list*)
  (without-interrupts
   (pop *global-list*))
```

```lisp
(without-interrupts
  (mp:push value *global-list*)
  (mp:pop *global-list*)
)```
New: use low level atomic operations.

(sys:atomic-push value *global-list*)
(sys:atomic-pop *global-list*)

**19.8.4.3 Atomic push/delete**

Old:

(without-interrupts
 (push value *global-list*))
(without-interrupts
 (setq *global-list* (delete value *global-list*)))

New: use a lock, because delete cannot be done atomically since it reads more than one object before modifying one of them.

(defvar *global-list-lock* (mp:make-lock :name "Global List"))

(mp:with-lock (*global-list-lock*)
 (push value *global-list*))

(mp:with-lock (*global-list-lock*)
 (setq *global-list* (delete value *global-list*)))

**19.8.4.4 Atomic plist update**

Old:

(without-interrupts
 (setf (getf *global-plist* key) value))
(without-interrupts
 (getf *global-plist* key))

New: use a lock, because a plist consists of more than one object so cannot be updated with low level atomic operations.

(defvar *global-plist-lock* (mp:make-lock :name "Global Plist"))

(mp:with-lock (*global-plist-lock*)
 (setf (getf *global-plist* key) value))

(mp:with-lock (*global-plist-lock*)
 (getf *global-plist* key))
19.8.4.5 Atomic update of a data structure

The example below is a resource object, which maintains a count of free items and also list of them. These two slots must stay synchronized.

Old:

```lisp
(without-interrupts
  (when (plusp (resource-free-item-count resource))
    (decf (resource-free-item-count resource))
    (pop (resource-free-items resource)))
```

New: use a lock, because more than one slot has to be updated, so cannot be updated with low level atomic operations.

```lisp
(mp:with-lock ((resource-lock resource))
  (when (plusp (resource-free-item-count resource))
    (decf (resource-free-item-count resource))
    (pop (resource-free-items resource)))
```

19.8.4.6 Atomic access to a cache in a hash table

Old:

```lisp
(without-interrupts
  (or (gethash value *global-hashtable*)
    (setf (gethash value *global-hashtable*)
      (make-cached-value)))
```

New: use the hash table lock.

```lisp
(hcl:with-hash-table-locked *global-hashtable*
  (or (gethash value *global-hashtable*)
    (setf (gethash value *global-hashtable*)
      (make-cached-value)))
```

Alternative new: use the hash table lock only if the value is not already cached. This can be faster than the code above, because it avoids locking the hash table for concurrent reads.

```lisp
(or (gethash value *global-hashtable*); probe without the lock
  (hcl:with-hash-table-locked *global-hashtable*
    (or (gethash value *global-hashtable*); reread with the lock
      (setf (gethash value *global-hashtable*)
        (make-cached-value))))
```
19.9 Timers

Use timers to run code after a specified time has passed. You can schedule a timer to run once or repeat at regular intervals, and you can unschedule it before it expires.

The timers are measured in elapsed time and the accuracy depends on various factors, including the operating system and the load on the computer.

For the details, see the reference entries for `make-timer` and `schedule-timer`.

19.9.1 Timers and multiprocessing

Timers run in unpredictable threads, therefore it is not safe to run code that interacts with the user directly. The recommended solution is something like

```lisp
(mp:schedule-timer-relative
 (mp:make-timer 'capi:execute-with-interface
   interface
   'capi:display-message "Time's up")
  5)
```

or

```lisp
(mp:schedule-timer
 (mp:make-timer 'capi:execute-with-interface
   interface
   'capi:display-message "Lunchtime")
 (* 4 60 60))
```

where `interface` is an existing CAPI interface on the screen.

Timers actually run in the process that is current when the scheduled time is reached. This is likely to be The Idle Process in cases where LispWorks is sleeping, but it is inherently unpredictable.

19.9.2 Input and output for timer functions

I/O streams default to the standard input and output of the process, which is initially `*terminal-io*` in the case of The Idle Process.
19.10 Process properties

A "process property" is a pair of an indicator and a value that is associated with it for a process.

LispWorks has two kinds of process properties: general and private. These two kinds of properties are stored separately, and the association of indicator/value in each property kind is independent of any in the other property kind.

General properties are stored in the process plist, and can be modified from other processes.

Private properties can only be modified by the current process. Private properties are faster to modify, because the modification does not need to be thread-safe.

Otherwise there is little difference between general and private properties.

process-plist and (setf process-plist) are not thread-safe. In LispWorks 5.1 and earlier the only interface to process properties is process-plist, but this does not work well in SMP LispWorks, and so it is deprecated.

There is no parallel to process-plist for the private properties.

The general properties are accessed by: process-property, (setf process-property), remove-process-property, pushnew-to-process-property and remove-from-process-property.

The private properties are accessed by: get-process-private-property (access from other processes), process-private-property, (setf process-private-property), remove-process-private-property, pushnew-to-process-private-property and remove-from-process-private-property.

19.11 Other processes functions

19.11.1 Process Priorities

Each process has a priority and can either be runnable, blocked or suspended.

The effect of process priorities is significantly different between SMP LispWorks and non-SMP LispWorks.
19.11 Other processes functions

19.11.1 Process priorities in SMP LispWorks

Process priorities are almost completely ignored in SMP LispWorks. The main exception is that for processes that wait with `process-wait` for something to happen, a process with higher priority is likely to wake up earlier, but even then it is not guaranteed.

19.11.1.2 Process priorities in non-SMP LispWorks

If there is a runnable process with priority \( P \), then no processes with priority less than \( P \) will run. When there are runnable processes with equal priority, they will be scheduled in a round-robin manner.

If a process with priority \( P \) is running and a blocked process with priority greater than \( P \) becomes runnable, the second process will run when the scheduler is next invoked (either explicitly or at the next preemption tick).

To find the priority of a process, use `process-priority`. This can be changed using `change-process-priority`.

\[
\text{(mp:change-process-priority proc-1 10)}
\]

Another way to specify the priority is to create the process with `process-run-function`, passing the keyword `:priority`:

\[
\text{(list}
\text{ (mp:process-run-function}
\text{ "SORTER-DOT" '(:priority 10) #'sorter #\.}
\text{ (mp:process-run-function}
\text{ "SORTER-DASH" () #'sorter #\-))}
\]

19.11.2 Accessing symbol values across processes

Use `symeval-in-process` to read the value of a dynamically bound symbol in a given process.

\[
\text{(setf mp:symeval-in-process)}
\]

`symeval-in-process` is mostly intended for debugging. Do not call it while the thread is actually running.
19.11.3 Stopping and unstopping processes

This section describes a typical way of using `process-stop` and `process-unstop`.

Suppose a pool of "worker" processes is managed by a "manager" process. A process in the worker pool marks itself as available for work, and then calls `process-stop`. The manager process later finds a worker process that is marked as available for work, puts the work in a place known to the worker process, and then calls `process-unstop` on the worker process.

For this scheme to work properly, the check of whether the worker is available needs to include a call to `process-stopped-p`. Otherwise, it is possible for the following sequence of events to occur:

1. A worker marks itself as available.
2. The manager process finds the worker and gives it the work.
3. The manager process calls `process-unstop` on the worker.
4. The worker process proceeds and calls `process-stop`, and never wakes up.

To guard against this possibility, then the manager should call `process-stopped-p` when finding the worker in the second step above. Alternatively, it could check the result of `process-unstop`.

19.12 Native threads and foreign code

Support for native threads differs between platforms as described in this section.

19.12.1 Native threads on Windows, Mac OS X, Linux, x86/x64 Solaris, FreeBSD and AIX

Each Lisp `mp:process` has a separate native thread and in LispWorks 6.0 and later versions these threads can run simultaneously.

Note: In LispWorks 5.1 and earlier versions, you can have many runnable `mp:process` objects/native threads, but Lisp code can only run in one thread at a time and a lock is used to enforce this. This can limit performance on a
Native threads and foreign code

19.12 Native threads and foreign code

When a foreign function is called using the FLI, the lock is released until the function returns. This allows other Lisp threads to run, for instance while waiting for a database query to execute.

You can call back into Lisp using fli:define-foreign-callable in any thread, without any other setup.

Threads running Lisp code can be rescheduled preemptively, so if you call into Lisp from more than one thread simultaneously and one request takes a long time then it will not delay the requests in other threads.

19.12.2 Native threads on other platforms

Lisp uses a single native thread and implements user level threads to support mp:process.

You can only call back into Lisp from its single native thread.

Note: This section applies only to LispWorks for SPARC Solaris.

19.12.3 Foreign callbacks on threads not created by Lisp

When foreign code creates a native thread (a “foreign thread”) and code running on this thread calls into Lisp, then Lisp needs to associate a Lisp process object with this thread to be able to work properly.

When there is a call on a foreign thread into Lisp which is not a recursive call (an "outer call"), Lisp first checks if there is a process associated with this thread, and if there is it uses it. Otherwise, it creates a new process and associates it with the foreign thread. Recursive calls into Lisp (when Lisp calls foreign code which calls back into Lisp) are processed in the same way as recursive calls in Lisp threads.

When the outer call returns, Lisp by default keeps the process associated with the thread, but this is not guaranteed. Keeping the process means that next call into Lisp requires less work, but comes at the cost of using more memory. Lisp eliminates the process if it detects that the thread has died, if there is call to last-callback-on-thread inside the outer call or if the process is killed by process-terminate.
Once Lisp has a process associated with the thread, it establishes it as the current process, as returned by calling \texttt{get-current-process}, and then calls the foreign callable Lisp code.

Part of establishing the process involves binding the variables in \texttt{*process-initial-bindings*}. Note that this binding happen repeatedly for each outer call. The computation of the bound value is done when the process is created, so if the process is not eliminated between outer calls (the default behavior), this happens only once. The computation of the value occurs in the dynamic environment of the new process.

Compatibility note: Before LispWorks 7.1, the computation occurred in a "no-process" scope, and an error would have entered the debugger in the console without an option to abort.

\textbf{19.12.3.1 Performance considerations for foreign threads}

Keeping the process between outer calls (the default behavior), makes each call faster, but uses memory. For few processes, this is probably the best approach. With many processes, the memory usage may become an issue.

There is an overhead for an outer call, which is larger than a recursive call. A few outer calls per second should not be a problem, but it should be avoided inside a heavy computation.
19.13 Low level operations

19.13.1 Low level atomic operations

Low level atomic operations are defined in all cases for a specific set of places. These places are listed in Table 19.1:

Table 19.1 Places for low-level atomic operations

<table>
<thead>
<tr>
<th>Place</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(symbol-value symbol)</code></td>
<td>When <code>symbol</code> is dynamically bound, this means the dynamically bound value.</td>
</tr>
<tr>
<td>A dynamically bound <code>symbol</code></td>
<td>The dynamically bound value.</td>
</tr>
<tr>
<td>A lexically bound <code>symbol</code></td>
<td>It is an error to use a low level atomic operation on a lexically bound symbol.</td>
</tr>
<tr>
<td><code>(car cons)</code></td>
<td></td>
</tr>
<tr>
<td><code>(cdr cons)</code></td>
<td></td>
</tr>
<tr>
<td><code>(the type place)</code></td>
<td>For another <code>place</code> listed in this table.</td>
</tr>
<tr>
<td><code>(svref sv index)</code></td>
<td>Only <code>simple-vector</code>.</td>
</tr>
<tr>
<td>Structure accessors</td>
<td>The structure must be defined at compile time.</td>
</tr>
<tr>
<td><code>(slot-value object slot-name)</code></td>
<td>See below.</td>
</tr>
</tbody>
</table>

Notes about atomic `slot-value` operations:

1. They ignore the MOP `slot-value-using-class` protocol and can only be used for `:instance` and `:class` allocated slots.
2. They are slower than the atomic operations on other types of object because they have to lock the instance. Normally it would be better to have a slot pointing to some other object (for example a structure) and do the atomic operations on that object.

The low level atomic operations implicitly ensure order of memory between operations in different threads.

The low level atomic operations are: `atomic-push, atomic-pop, atomic-fixnum-incf, atomic-fixnum-decf, atomic-incf, atomic-decf, atomic-exchange` and `compare-and-swap`.

Application of macros that are defined by `define-atomic-modify-macro` is also restricted to the places in Table 19.1 above, because they implicitly use low level atomic operations.

You can test whether a place is suitable for use with these operations by the predicate `low-level-atomic-place-p`.

### 19.13.2 Aids for implementing modification checks

The macros `with-modification-check-macro` and `with-modification-change` provide a way for a body of code to execute and check whether there was any "modification" during this execution, where modification is execution of some other piece of code. This is useful in situations when reading some data out of some data structure is more common than modification, and reading the data involves getting some values that need to be consistent. It makes it possible to ensure consistency of the values without a lock.

The checking code should be wrapped by the macro `with-modification-check-macro`, and the modifying code should be wrapped by the macro `with-modification-change`. They are associated by the fact that their `modification-place` argument is the same.

`modification-place` is a place as defined in Common Lisp (it does not need to be one of the places for atomic locking) which can receive a fixnum. It must be initialized to a fixnum. It must not be modified by any code except `with-modification-change`.  

---

300
with-modification-check-macro defines a lexical macro (by macrolet) with the name macro-name which takes no arguments, and is used to check whether there was any change since the entering the body.

Note that these macros do not guard against errors that may occur because of changes to the data structures that are accessed, and do not create any locking between users of these macro. In particular, the modifying code will typically need to lock something too, and the checking code must do only operations that cannot fail because of modification in another thread.

### 19.13.2.1 Example modification check

```lisp
(defstruct my-cache
    (modification-count 0) a b)

;; modifier code
(sys:with-modification-change
    (my-cache-modification-count cache)
    (setf (my-cache-a cache) (calculate-a-value ....)
          (my-cache-b cache) (calculate-b-value .....)))

;; reading code
(loop
    (sys:with-modification-check-macro
        my-cache-did-not-change-p (my-cache-modification-count cache)
        (let ((a (my-cache-a cache))
                (b (my-cache-b cache)))
            (when (my-cache-did-not-change-p)
                (return (values a b))))))
```

Provided that all modification to the `a` and `b` slots of a `my-cache` object are done by the modifier code above, the return values of `a` and `b` in the reading code are guaranteed to have been set by the same `setf` invocation in the modifier code.

### 19.13.3 Ensuring order of memory between operations in different threads

A set of synchronization functions is provided which ensure order of memory between operations in different threads. These are `ensure-loads-after-`

Note: You should have a good understanding of multiprocessing issues at the CPU level to write code that actually needs these functions.

The effect of each of these functions is to ensure that all the operations of the first type (the word following the ensure-) that are in the program after the call to the function are executed after all the operations of the second type (last word in the function name) that are in the program before the call to the function.

Before or after "in the program" means the order that a programmer interpreting (correctly) the program would expect the operations to be executed. On a modern CPU this is not necessarily the same as the actual execution order. On a single CPU the end result is guaranteed to be the same, but on a computer with multiple CPU cores it is not.

An operation of type load is an operation that reads data from an object into a local variable. Typical load operations are car, cdr, svref, structure accessors, slot-value and getting the value of a symbol. A store operation is an operation that modifies data in an object. A memory operation is either a load or a store.

You need these functions when you need to synchronize between threads and you do not want to use the system supplied synchronization objects ("Locks", mailboxes, "Condition variables", "Counting semaphores", "Synchronization barriers"). In most cases you should try first to use a synchronization object. Using the synchronization functions described in this section is useful if you can identify a serious bottleneck in your code that can be optimized using them.

For simple cases you should consider whether with-modification-check-macro and with-modification-change gives you the functionality you need.

19.13.3.1 Example of ensuring order of memory

Suppose you have two code fragments, which may end up executed in parallel, and both of which access a global structure *gs*. The first fragment is a
setter, and you can be sure that it is not executed in parallel to itself (normally because it actually runs while holding a lock):

```lisp
(setf (my-structure-value-slot *gs*) ; store1 some-value)
(setf (my-structure-counter-slot *gs*) ; store2 counter)
```

The second fragment is the reader. You want to guarantee that it gets a value that was stored after the counter reached some value (the counter value always increases). You may think that this will suffice:

```lisp
(if (>= (my-structure-counter-slot *gs*) ; load1 counter)
    (my-structure-value-slot *gs*)    ; load2
    (.. something else ...))
```

Programmatically, if the >= is true then store2 already occurred before load1, therefore store1 also occurred before load1, and load2 which happens after load1 must happen after store1.

On a single CPU that is true. On a computer with multiple CPU cores it can go wrong (that is, load2 can happen before store1) because of two possible reasons:

1. load2 may happen before load1.
2. store2 may happen before store1.

To guarantee that load2 happens after store1, both of these possibilities need to be dealt with. Thus the setter has to be:

```lisp
(setf (my-structure-value-slot *gs*) ; store1 some-value)
(sys:ensure-stores-after-stores) ; ensure store order
(setf (my-structure-counter-slot *gs*) ; store2 incf-counter))
```

and the reader has to be:
19 Multiprocessing

```
(if (> (my-structure-counter-slot *gs*)) ; load1
  my-counter)
(progn
  (sys:ensure-loads-after-loads) ; ensure load order
  (my-structure-value-slot *gs*)) ; load2
(.. something else ...))
```

Note that somehow both threads know about counter, and normally will have to synchronize the getting of its value too.
This chapter describes tools for managing programs comprising many source files. Most of the material concerns LispWorks’ native defsystem ("Common Defsystem"), but the last section describes how to use the popular open source alternative ASDF with LispWorks.

20.1 Introduction

When an application becomes large, it is usually prudent to divide its source into separate files. This makes the individual parts of the program easier to find and speeds up editing and compiling. When you make a small change to one file, just recompiling that file may be all that is necessary to bring the whole program up to date.

The drawback of this approach is that it is difficult to keep track of many separate files of source code. If you want to load the whole program from scratch, you need to load several files, which is tedious to do manually, as well as prone to error. Similarly, if you wish to recompile the whole program, you must check every file in the program to see if the source file is out of date with respect to the object file, and if so re-compile it.

To make matters more complicated, files often have interdependencies; files containing macros must be loaded before files that use them are compiled. Similarly, compilation of one file may necessitate the compilation of another
file even if its object file is not out of date. Furthermore, one application may consist of files of more than one source code language, for example Lisp files and C files. This means that different compilation and loading mechanisms are required.

The LispWorks system tools, and the System Browser in the LispWorks IDE, are designed to take care of these problems, allowing consistent development and maintenance of large programs spread over many files. A system is basically a collection of files that together constitute a program (or a part of a program), plus rules expressing any interdependencies which exist between these files.

You can define a system in your source code using the `defsystem` macro. Once defined, operations such as loading, compiling and printing can be performed on the system as a whole. The system tools ensure that these operations are carried out completely and consistently, without doing unnecessary work. A system may itself have other systems as members, allowing a program to consist of a hierarchy of systems. Each system is treated independently of the others, and can be used to collect related pieces of code within the overall program. Operations on higher-level systems are invoked recursively on member systems.

It is also possible to define a system in your code using `asdf:defsystem`, an open source system definition utility with similar functionality to LispWorks `defsystem`. See “Using ASDF” on page 311 for a description of how to use ASDF with LispWorks.

### 20.2 Defining a system

A system is defined with a `defsystem` form in an ordinary Lisp source file. This form must be loaded into the Lisp image in order to define the system in the environment. Once loaded, operations can be carried out on the system by invoking Lisp functions, or, more conveniently, by using the system browser.

For example, the expression:

```
CL-USER 5 > (compile-system 'debug-app :force t)
```

would compile every file in a system called `debug-app`. 
20.2 Defining a system

**Note:** When defining a hierarchy of systems, the leaf systems must be defined first — that is, a system must be declared before any systems that include it.

By convention, system definitions are placed in a file called `defsys.lisp` which usually resides in the same directory as the members of the system.

The full syntax is given in `defsystem`, page 1109. Below is a brief introduction.

### 20.2.1 DEFSYSTEM syntax

`defsystem` takes four arguments: `name`, `options`, `members` and `rules`.

- **name** should be a string that names the system.
- **options** is a list of keyword-value pairs specifying attributes of the system such as the default location of its member files or the default compiler optimize qualities in effect when `compile-system` is called.
- **members** lists the members of the system which can be source files (of Common Lisp or foreign code) or other systems (that is, subsystems).
- **rules** is a set of rules describing the requirements for compilation and loading of the system members and the order in which this should take place.

See the following sections for more information about these parameters.

### 20.2.2 DEFSYSTEM options

Options may be specified to `defsystem` which affect the behavior of the system as a whole. For example, `:package` specifies a default package into which files in the system are compiled and loaded if the file itself does not contain its own package declaration. The `:defaultpathname` option tells the system tools where to find files which are not expressed as a full pathname.

### 20.2.3 DEFSYSTEM members

The `:members` keyword to `defsystem` is used to specify the members of a system. The argument given to `:members` is a list of strings. A system member is either a file or another system, identified by a name. If a full pathname is given then the function `pathname-name` is used to identify the name of the
member. Thus, for example, the name of a member expressed as
/u/dubya/foo.lisp is foo.

System members must have unique names, by a case-insensitive string comparison, so if a system has a member called "foo" then it cannot have another member (a file or a system) named "foo", "FOO" or foo.

The behavior of any member within a system can be constrained by supplying keyword arguments to the member itself. So, for example, specifying the :source-only keyword ensures that only the source file for that member is ever loaded.

### 20.2.4 DEFSYSTEM rules

Rules may be defined in a system which modify the default behavior of that system, ensuring, for instance, that certain files are always loaded or compiled before others.

Rules apply to files and subsystems alike as members of their parent system, but are not inherited by subsystems.

When you invoke an action such as compiling a system, the following happens by default:

- Each member of the system is considered in turn, in the order they are given in the system definition.
- If the member is itself a system then the action is performed on that system too, and so on recursively.
- If the member is a file and action-specific constraints are satisfied, the file action is inserted into a plan.
  
  For example, in the case of compiling, a “compile this file” event is put into the plan if the source file is newer than the object file.
- After the plan has been assembled, it can be viewed or executed.

This behavior can be modified by describing dependencies between the members using rules. These are specified using the :rules keyword to defsystem.

A rule has three components:
20.2 Defining a system

The target(s). The action that is performed if the rule executes successfully.
This is an action-member description like `:compile "foo"`. The member can be an actual member of the system or `:all` (meaning the rule should apply to each member of the system).

The actions that the target(s) are `:caused-by`.
The actions that cause the rule to execute successfully.
This is a list of action-member descriptions. The member of each of these descriptions should be either a real system member, or `:previous`, which means all members listed before the member of the target in the system description.
If any of these descriptions are already in the current plan (as a result of other rules executing successfully, or as a result of default system behavior), they trigger successful execution of this rule.

The actions that the target(s) `:requires`.
The actions that need to be performed before the rule can execute successfully.
This is a list of action-member descriptions that should be planned for before the action on the target(s). Again, each member should either be a real member of the system, or `:previous`.

The use of the keyword `:previous` means, for example, that you can specify that in order to compile a file in the system, all the members that come before it must be loaded.

When the action and member of a target are matched during the traversal of the list of members, the target is inserted into the plan if either of the following are true:

- any of the action-member descriptions in the `:caused-by` clause is already in the plan, or
any implicit conditions (such as the source file being newer than the object file) are satisfied.

If the target is put into the plan then other targets are inserted beforehand if the action-member description of any :requires clause is not already in the plan.

### 20.2.5 Examples

Consider an example system, `demo`, defined as follows:

```
(defsystem demo (:package "USER")
  :members ("parent" *child1* "child2")
  :rules ((:in-order-to :compile (*child1* *child2*)
             (:caused-by (compile "parent"))
             (:requires (load "parent")))
```

This system compiles and loads members into the `USER` package if the members themselves do not specify packages. The system contains three members — `parent`, `child1`, and `child2` — which may themselves be either files or other systems. There is only one explicit rule in the example. If `parent` needs to be compiled (for instance, if it has been changed), then this causes `child1` and `child2` to be compiled as well, irrespective of whether they have themselves changed. In order for them to be compiled, `parent` must first be loaded.

Implicitly, it is always the case that if any member changes, it needs to be compiled when you compile the system. The explicit rule above means that if the changed member happens to be `parent`, then every member gets compiled. If the changed member is not `parent`, then `parent` must at least be loaded before compiling takes place.

The next example shows a system consisting of three files:

```
(defsystem my-system
  (:default-pathname "-junk/")
  :members ("a" "b" "c")
  :rules ((:in-order-to :compile (*c*)
             (:requires (load "a"))
             (:caused-by (compile "b"))))
```

What plan is produced when all three files have already been compiled, but the file \texttt{b.lisp} has since been changed?

First, file \texttt{a.lisp} is considered. This file has already been compiled, so no instructions are added to the plan.

Second, file \texttt{b.lisp} is considered. Since this file has changed, the instruction \texttt{compile b} is added to the plan.

Finally file \texttt{c.lisp} is considered. Although this has already been compiled, the clause

\begin{verbatim}
(:caused-by (:compile "b"))
\end{verbatim}

causes the instruction \texttt{compile c} to be added to the plan. The compilation of \texttt{c.lisp} also requires that \texttt{a.lisp} is loaded, so the instruction \texttt{load a} is added to the plan first. This gives us the following plan:

1. Compile \texttt{b.lisp}.
2. Load \texttt{a.lisp}.
3. Compile \texttt{c.lisp}.

This last example shows how to make each fasl get loaded immediately after compiling it:

\begin{verbatim}
(defun my-system ()
  (members "foo" "bar" "baz" "quux")
  (rules ((:in-order-to :compile :all
             (:requires (:load :previous))))))

(compile-system my-system :load t)
\end{verbatim}

\section{Using ASDF}

You can load the supplied version of ASDF 2 by

\begin{verbatim}
(require "asdf")
\end{verbatim}

Optionally, if you actually want your later version of ASDF 2, do

\begin{verbatim}
(asdf:load-system :asdf)
\end{verbatim}
You may need to configure ASDF. For the language-level interface you should follow the ASDF documentation at http://common-lisp.net/project/asdf/.

Then load your ASDF system definitions and you are ready to work with ASDF systems in LispWorks.

It is possible to work with both Common Defsystem and ASDF in the same LispWorks image, as long as you use the appropriate APIs to operate on each type of system.

### 20.3.1 Bypassing the supplied version of ASDF

To use a specific version of ASDF 2 without loading the version supplied with LispWorks, you should load it directly and then call

```lisp
(provide *asdf*)
```

to prevent the distributed version from being loaded later.

### 20.3.2 Using ASDF in the LispWorks IDE

You can work with your ASDF systems using the LispWorks IDE tools.

This needs some integration code which makes the System Browser, Editor and Search Files tools work with ASDF systems as well as 'native' LispWorks systems. The ASDF integration code is in

```lisp
(example-edit-file "misc/asdf-integration")
```

in the LispWorks library and if necessary you can load it directly. However, it is more convenient to rely on this code being loaded automatically.

The variable `*autoload-asdf-integration*` is consulted when the LispWorks IDE starts. If its value is true (this is the default) then the ASDF integration code is loaded automatically when ASDF is loaded.

See the comments in `asdf-integration.lisp` for more information about using ASDF with LispWorks.
21

The Parser Generator

21.1 Introduction

The parser generator generates an LALR parser from a specification of a grammar. The parser generator has a simple facility for the static resolution of ambiguity in the grammar and supports an automatic run time error correction mechanism as well as user-defined error correction. Semantic actions can be included in the rules for the grammar by specifying Lisp forms to be evaluated when reductions are performed.

For further details on LALR parsing, see Compilers, Principles Techniques and Tools, by Aho, Sethi and Ullman, publishers Addison Wesley, 1986.

Load the parser generator by `(require "parsergen")`.

21.2 Grammar rules

The parser generator is accessed by the macro `defparser`. After the `name`, of the parser, the macro form specifies the reduction rules and semantic actions for the grammar.

The `rules` specified in a `defparser` form are of three types, normal rules, error rules and combined-rules, described below.

Each normal rule corresponds to one production of the grammar to be parsed:
The non-terminal is the left-hand side of the grammar production and the list of grammar symbols defines the right-hand side of the production. (The right-hand side may be empty.) The list of forms specifies the semantic action to be taken when the reduction is made by the parser. These forms may contain references to the variables $1 \ldots n$, where $n$ is the length of the right hand side of the production. When the reduction is done, these variables are bound to the semantic values corresponding to the grammar symbols of the rule.

### 21.2.1 Example

If a grammar contains the production:

```
expression --> expression operator expression
```

with a semantic representation of a list of the individual semantic values, the Lisp grammar would contain the rule:

```
((expression expression operator expression) (list $1 $2 $3))
```

Error productions of the form:

```
((nt :error) (some error behavior))
```

are explained in the section below.

The first rule of the grammar should be of the form:

```
((nt ntl) $1)
```

where the non-terminal nt has no other productions and ntl serves as the main “top-level” non-terminal.

### 21.2.2 Combined rules

The combined-rule clause is a way to group multiple normal-rule or error-rule clauses for the same non-terminal. For example, this single combined-rule clause:

```
(constant
  ((:FLOAT-CONSTANT) $1)
  ((:INTEGER-CONSTANT) $1))
```
is equivalent to these two normal-rule clauses:

```lisp
((constant :FLOAT-CONSTANT) $1)
((constant :INTEGER-CONSTANT) $1)
```

### 21.2.3 Resolving ambiguities

If the grammar is ambiguous, there is conflict between rules of the grammar: either between reducing with two different rules or between reducing by a rule and shifting an input symbol. Such a conflict is resolved at parser generation time by selecting the highest priority action, where the priority of a reduce action is determined by the closeness of the rule to the beginning of the grammar. A priority is assigned to a shift by associating it with the rule that results in the shift being performed.

For example, if the grammar contains the two rules:

- Rule a: `statement -> :if expression :then statement :else statement`
- Rule b: `statement -> :if expression :then statement`

this results in a conflict in the parser between a shift of `:else`, for rule a, and a reduce by rule b. This conflict may be resolved by listing rule a earlier in the grammar than rule b. This ensures that the shift is always done.

Note that ambiguities cannot always be resolved successfully in this way. In this example, if the ambiguity is resolved the other way around, by listing rule b first, this results in the `if ... then ...` part of an `if ... then ... else ...` statement being reduced, and a syntax error is produced for the `else` part.

During parser generation, any conflicts between rules are reported, together with information about how the conflict was resolved.

### 21.3 Functions defined by defparser

The form `(defparser name grammar)` defines a number of functions. The main function `name` is defined as the parsing function. For example:

```lisp
(defparser my-parser .. grammar ..)
```

defines the function

```lisp
my-parser lexer &optional symbol-to-string =>
```
lexer specifies the lexical analyzer function to be used. The optional argument symbol-to-string should be a function mapping grammar symbols to strings for printing purposes. The default value of symbol-to-string is the function cl:identity.

defparser also defines functions corresponding to the individual actions of the parser.

Normal actions are named:

\texttt{name-actionindex}

and error actions are named:

\texttt{name-error-actionindex}

where name here is the name as given to defparser and index is the number of the rule or error rule in the grammar.

All function names are interned in the current package when defparser is called.

### 21.4 Error handling

The parser supports automatic error correction of its input. The strategy used involves attempting to either push a new token onto the input, replacing an erroneous symbol, or discarding an erroneous symbol. Such action is only taken if it is guaranteed that the parser can continue parsing and read at least one more symbol from its input.

If the correction strategy fails, then error recovery is invoked.

The parser allows the inclusion of grammar productions of the form:

\texttt{non-terminal \rightarrow :error}

This means that the parser accepts an erroneous string of tokens as constituting an occurrence of the non-terminal. Such productions may be used to skip over portions of input when attempting to recover from an error. The action associated with such an error is specified by a form in the same way as for ordinary actions. The action may perform manipulation of the parser state and input.
21.5 Interface to lexical analyzer

The lexical analyzer function that is passed to the parser is expected to be a function of zero arguments that returns two values each time it is called. The first value is the next token on the input and the second value is the semantic value corresponding to that token. If there is no more input, then the lexical analyzer may return either the token `:eoi` or `nil`.

For example:

```lisp
(defparser my-parser
  ...)

(defun my-lexer (stream)
  ;; read next token from stream ..
  (values token value))
(defun my-symbol-to-string (symbol)
  ;; returns a string ..)
(defun my-parse-stream (stream)
  (let ((lexer #'(lambda () (my-lexer stream))))
    (my-parser lexer #'my-symbol-to-string)))
```

Note that during error correction, the parser may push extra tokens onto the input, in which case they are given the semantic value `nil`. The semantic actions should therefore be capable of dealing with this situation. Manipulation of the input (for example pushing extra tokens) is done within the parser generator and the lexical analyzer need not concern itself with this.

21.6 Example

The following example shows a simple grammar for a very small subset of English.
(defpackage "ENGLISH-PARSER"
  (:use *PARSERGEN*)
  (:add-use-defaults t))

(in-package "ENGLISH-PARSER")

;;; Define the parser itself.

(defparser english-parser
  ((bs s) $1)
  ((s np vp) 
     `($1 ,$2))
  ((bnp :adj bnp) 
     `($1 ,$2))
  ((bnp bnp relp) 
     `($1 ,$2))
  ((bnp :noun) $1)
  ((relp :rel vp) 
     `($1 ,$2))
  ((vp :verb np locp) 
     `($1 ,$2 ,$3))
  ((vp :verb locp) 
     `($1 ,$2))
  ((vp :verb np) 
     `($1 ,$2))
  ((vp :verb) 
     $1)
  ((np :art bnp locp) 
     `($1 ,$2 ,$3))
  ((np :art bnp) 
     `($1 ,$2))
  ((np bnp) $1)
  ((locp :loc np) 
     `($1 ,$2))))

;;; The lexer function.

;;; The basic lexing function

(defvar *input*)
(defun lex-english ()
  (let ((symbol (pop *input*)))
    (if symbol (get-lex-class symbol) nil)))

;;; Getting syntactic categories.

(defparameter *words*

(defun get-lex-class (word)
  (values
    (or (cadr (assoc word *words*)) :unknown)
    word))

;;; The main function -- note bindings of globals (these ;;; are exported from the parsergen package).

(defun parse-english (input)
  (let ((*input* input))
    (english-parser #'lex-english)))

The following example session shows the parsing of some sentences.

ENGLISH-PARSER 34 > (parse-english ' (the cat sat on the
mat))
((THE CAT) (SAT (ON (THE MAT))))

ENGLISH-PARSER 35 > (parse-english ' (the big brown dog
behind the door ate the cat
which sat on the floor))
((THE (BIG (BROWN DOG)) (BEHIND (THE DOOR)))
 (ATE (THE (CAT (WHICH (SAT (ON (THE FLOOR))))))))
21 The Parser Generator
Dynamic Data Exchange

22.1 Introduction

Dynamic data exchange (DDE) involves passing data and instructions between applications running under the Microsoft Windows operating system. Typically the data is passed in the form of a string, which is interpreted when it is received. One application acts as a server and the other as a client.

22.1.1 Types of transaction

The server is normally a passive object, which waits for a client object to tell it what to do. The client can communicate with the server in four ways:

- The client can issue a request transaction to the server. This means the client is asking for some information about the server application.
- The client can issue a poke transaction. This means the client is passing data to be stored by the server application.
- The client can issue an execute transaction. This means the client is asking the server to get the server application to run a command.
- The client can ask the service to set up an advise loop, or to close an existing advise loop. An advise loop causes the server to communicate with the client whenever a specified change occurs in the server application.
22.1.2 Conversations, servers, topics, and items

For a transaction to take place between a client and a server, a conversation must be established. A conversation is established when a client makes a request by broadcasting a service name and topic name, and a server responds. Transactions can then take place across the conversation. When no more transactions are to be made, the conversation is terminated.

The following list identifies the elements involved with client/server activity:

- **conversation**: A conversation is established when a server responds to a client.
- **service name**: A service name is a string broadcast by a client hoping to establish a conversation with a server that recognizes the service name. The service name is usually clearly related to the server application name.
- **topic name**: The topic name identifies what the conversation between client and server is to be about. For example, it could be the name of a file that is open in the server application. Each topic is attached to one particular server. A server can have many topics.
- **item name**: The item usually identifies an element of the file identified by the topic which should be read (in the case of a request) or written to (in the case of a poke). For example, it might refer to a cell in a spreadsheet document.

22.1.3 Advise loops

A DDE advise loop describes a connection back to the application that is used to track changes to a DDE topic. It instructs the server to inform the client when data in the server’s application changes. Advise loops are set up across a conversation, and closing the conversation closes the advise loop.

An advise loop is identified by an item and a key. The key is included to allow any number of uniquely identifiable advise loops to be set up on the same server/topic/item combination.

A successfully established advise loop is also known as a link. When a change occurs to item, the link informs the client by causing it to execute a function.
There are two types of link: the warm link which only informs the client that a change to item has occurred, and the hot link which also sends the new data across.

**Note:** a DDE advise loop is not a loop in the program source code. In particular it should not be confused with the "event loop" which is a loop in source code that processes low level events.

### 22.1.4 Execute transactions

When a client issues an execute transaction to a server, the command to be executed is transferred as a string. This involves the marshalling of the command and its arguments into a suitable string format. The standard format of such a string is:

```
[command(arg1,arg2,...)]
```

### 22.2 Client interface

#### 22.2.1 Opening and closing conversations

A LispWorks client can open a conversation by using `dde-connect`, which takes a service designator and a topic designator as its arguments. If successful, a conversation object is returned which can be used to refer to the conversation. Conversations are closed by the LispWorks client at the end of a transaction by using `dde-disconnect`.

Another method for managing conversations uses `with-dde-conversation` to bind a conversation with a server across a body of code. If no conversation is available for `with-dde-conversation`, then one is automatically opened. The code is executed and the conversation is closed after the body of code exits.

#### 22.2.2 Automatically managed conversations

There is an alternative to manually establishing a conversation and then disconnecting it once all transactions between server and client are concluded: the automatically managed conversation. Client functions that end with a *conductor automatically managed conversations.
A function handling an automatically managed conversation takes a service
designator and topic designator as two of its arguments, and either automatic-
cally establishes a conversation with a server responding to the service desig-
nator/topic designator pair, or uses an existing equivalent conversation. For
the purpose of brevity, functions conducting automatically managed conver-
sations are only briefly mentioned in this chapter. For the details see `dde-
advise-start*`, `dde-advise-stop*`, `dde-execute*`, `dde-execute-com-
mand*`, `dde-execute-string*`, `dde-item*`, `dde-poke*` and `dde-request*`.

### 22.2.3 Advise loops

A LispWorks client can set up an advise loop across a conversation using `dde-
advise-start`, which takes a conversation (or a service designator/topic desig-
nator pair in the case of an automatically managed conversation using `dde-
advise-start*`), an item, and a key as its main arguments. The key argument
defaults to the conversation name, and can be used to distinguish between
multiple advise loops established on the same service/topic/item group.

Whenever the data monitored by the advise loop changes, a function is called
to inform the client. By default this function is the generic function `dde-cli-
ent-advise-data`. You can add methods to `dde-client-advise-data` spe-
cialized on the key or the client conversation class. Alternatively, you can
supply a different function in the call to `dde-advise-start`.

**Note:** a DDE advise loop is not a loop in the program source code. In particu-
lar it should not be confused with the "event loop" which is a loop in source
code that processes low level events.

#### 22.2.3.1 Example advise loop

The example shows you how to set up an advise loop. The code assumes that
`win32` package symbols are visible.

The first step defines a client conversation class, called `my-conv`.

```lisp
(defclass my-conv (dde-client-conversation) ()
)```
The macro `define-dde-client` can now be used to define a specific instance of the `my-conv` class for referring to a server application that responds to the service name "FOO".

```
(define-dde-client :foo :service "FOO" :class my-conv)
```

The next step defines a method on `dde-client-advise-data` which returns a string stating that the item has changed.

```
(defmethod dde-client-advise-data ((self my-conv) item data &key &allow-other-keys)
  (format t "~&Item ~s changed to ~s~%" item data))
```

Finally, the next command starts the advise loop on the server `foo`, with the topic name "file1", to monitor the item "slot1".

```
(dde-advise-start* :foo "file1" "slot1")
```

When the value of the item specified by "slot1" changes, the server calls `dde-client-advise-data` which returns a string, as described above.

The function argument of `dde-advise-start` and `dde-advise-start*` specifies the function called by the advise loop when it notices a change to the item it is monitoring. The function is `dde-client-advise-data` by default. A different function can be provided, and should have a lambda list similar to the following:

```
key item data &key conversation &allow-other-keys
```

The arguments `key` and `item` identify the advise loop, or link. The argument `data` contains the new data for hot links; for warm links it is `nil`.

Advise loops are closed using `dde-advise-stop` or `dde-advise-stop*`.

### 22.2.4 Request and poke transactions

LispWorks clients can issue request and poke transactions across a conversation using `dde-request` and `dde-poke`, which take a `conversation` (or a `service` designator/topic designator pair in the case of an automatically managed conversation), and an `item` as their main arguments. In the case of a poke transaction, data to be poked into `item` must also be provided.
In the case of a successful request transaction with \texttt{dde-request} or \texttt{dde-request*}, the data contained in \textit{item} is returned to the LispWorks client by the server.

In the case of a successful poke transaction with \texttt{dde-poke} or \texttt{dde-poke*}, the data provided is poked into \textit{item} by the server.

The accessor \texttt{dde-item} (or \texttt{dde-item*} for automatically managed conversations) can perform request and poke transactions. It performs a request transaction when read, and a poke transaction when set.

\section*{22.2.5 Execute transactions}

A client can issue an execute transaction across a conversation, or in the case of an automatically established conversation, to a recognized server. There is no need to specify a topic, as an execute transaction instructs the server application to execute a command.

The command and its arguments are issued to the server in the form of a string in a standard format (see “Execute transactions” on page 323). LispWorks provides two ways of issuing an execute transaction, namely \texttt{dde-execute-string} and \texttt{dde-execute-command} (and the corresponding * functions that automatically manage conversations).

The following example shows how \texttt{dde-execute-string*} can issue a command to a server designated by \texttt{:excel} on the topic \texttt{:system}, in order to open a file called \texttt{foo.xls}:

\begin{verbatim}
(dde-execute-string* :excel :system "[open("foo.xls")]")
\end{verbatim}

The function \texttt{dde-execute-command} takes the command to issue, and its arguments, and marshals these into an appropriate string for you. The following example shows how \texttt{dde-execute-command*} can issue the same command as in the previous example:

\begin{verbatim}
(dde-execute-command* :excel :system 'open '("foo.xls"))
\end{verbatim}
22.3 Server interface

22.3.1 Starting a DDE server

To provide a LispWorks application with a DDE server, follow the following three steps.

1. Define a specialized Lisp DDE server class using `define-dde-server`. Here the server class is called `foo-server` and it has the service name "FOO":
   ```lisp
   (define-dde-server foo-server "FOO")
   ```

2. Provide the server class with the functionality it requires by specializing methods on it and/or using `define-dde-server-function`. Here the server function is `bar`, which takes a string as an argument, and prints this to the standard output. For convenience, the system topic is used, though usually it is better to define your own topic.
   ```lisp
   (define-dde-server-function (bar :topic :system)
   :execute
   ((x string))
   (format t "~&~s~%" x)
   t)
   ```

   ```lisp
   (start-dde-server 'foo-server)
   ```
   This function returns the server object, which responds to requests for conversations with the service name "FOO", and accepts execute transactions for the function `bar` in the "System" topic.

22.3.2 Handling poke and request transactions

Poke and request transactions issued to a server object are handled by defining a method on each of the generic functions `dde-server-poke` and `dde-server-request`.

22.3.3 Topics

DDE servers respond to connection requests containing a service name and a topic name. The service name of a server is the same for any conversation
whereas the topic name may vary from conversation to conversation, and identifies the context of the conversation. Typically, valid topics correspond to open documents within the application, so the set of valid topics varies from time to time. In addition, all servers implement a topic called "System", which contains a standard set of items that can be read.

The LispWorks DDE interface supports three types of topics:

1. **General topics**
   A general topic is an instance of a user-defined topic class. The actual set of topics available may vary from time to time as the application is running.

2. **Dispatching topics**
   A dispatching topic has a fixed name, and is available at all times that the server is running. It supports a fixed set of items, and each of these items has Lisp code associated with it to implement these items.

3. **The system topic.**
   The system topic is provided automatically by the LispWorks DDE interface. However, a mechanism is provided to extend the functionality of the system topic by handling additional items.

### 22.3.3.1 General topics

To use general topics, the LispWorks application must define one or more subclasses of `dde-topic`. If an application supports only a single type of document, it will typically require only one topic class. If several different types of document are supported, it may be convenient to define a different topic class for each type of document.

If the application uses general topics, it should define a method on the `dde-server-topics` generic function, specializing on the application’s server class.

### 22.3.3.2 Dispatching topics

A dispatching topic is a topic which has a fixed name and always exists. Dispatching topics provide dispatching capabilities, whereby appropriate appli-
cation-supplied code is executed for each supported transaction. Dispatch topics are defined using `define-dde-dispatch-topic`.

### 22.3.3.3 The system topic

The system topic is implemented as a predefined dispatching topic called `:system`. It is automatically available to all defined DDE servers. Its class is `dde-system-topic`, which is a subclass of `dde-topic`.

The following items are implemented by the system topic:

#### SZDDESYS_ITEM_TOPICS

*Constant*

The constant `SZDDESYS_ITEM_TOPICS` has the value "Topics". Referring to this item in the system topic calls `dde-server-topics` to obtain a list of topics implemented by the server. The server should define a method on this generic function to return a list of strings naming the topics supported by the server. If this item is not to be implemented, do not define a method on the function, or define a method that returns `:unknown`.

#### SZDDESYS_ITEM_SYSITEMS

*Constant*

The constant `SZDDESYS_ITEM_SYSITEMS` has the value "SysItems". Referring to this item in the system topic calls `dde-topic-items` to obtain a list of items implemented by the system topic. If a server implements additional system topic items it should define a method on the generic function specialized on its server class and `dde-system-topic` returning the complete list of supported topics. The server can return `:unknown` if this item is not to be implemented.

#### SZDDESYS_ITEM_FORMATS

*Constant*

The constant `SZDDESYS_ITEM_FORMATS` has the value "Formats", and returns `unicodetext` and `text`. Currently only text formats are supported.

The system topic is a single object which is used by all DDE servers running in the Lisp image. You should therefore not under normal circumstances modify
it with `define-dde-server-function` by specifying a value of `:system` for the `topic` argument, as this would make the changes to the system topic visible to all users of DDE within the Lisp image.

Instead, specify `:server my-server :topic :system`, where `my-server` is the name of your DDE server. This makes the additional items available only on the system topic of the specified server.
This chapter is applicable to UNIX LispWorks and the Enterprise Edition of LispWorks. It describes Common SQL — the LispWorks interface to SQL. It should be used in conjunction with Chapter 47, “The SQL Package”, which contains full reference entries for all the symbols in the SQL package.

For a longer introduction to Common SQL, please see the SQL Tutorial available at [www.lispworks.com](http://www.lispworks.com).

### 23.1 Introduction

This chapter covers the following areas:

- Initialization and Connection
- The Functional SQL Interface
- The Object-Oriented (CLOS) SQL Interface
- The Symbolic SQL Syntax
- SQL I/O Recording
- SQL Interface Errors

The LispWorks SQL interface uses the following database terminology:
Data Definition Language (DDL)
The language used to specify and interrogate the structure of the database schema.

Data Manipulation Language (DML)
The language used for retrieving and modifying data. Also known as query language.

table
A set of records. Also known as relation.

attribute
A field of information in the table. Also known as column.

record
A complete set of attribute values in the table. Also known as tuple, or row.

view
A display of a table configured to your own needs. Also known as virtual table.

23.1.1 Overview
Common SQL is designed to provide both embedded and transparent access to relational databases from the LispWorks environment. That is, SQL/relational data can be directly manipulated from within Lisp, and also used as necessary when instantiating or accessing particular Lisp objects.

The SQL interface allows the following:
- Direct use of standard SQL statements as strings
- Mixed symbolic SQL and Common Lisp expressions
- Implicit SQL invocation when instantiating or accessing CLOS objects

The SQL interface provides these features through two complementary layers:
- A functional SQL interface
- An object-oriented SQL interface

The functional interface provides users with Lisp functions which map onto standard SQL DML and DDL commands. Special iteration constructs which utilize these functions are also provided. The object-oriented interface allows users to manipulate database views as CLOS classes via def-view-class.
23.1 Introduction

The two interfaces may be flexibly combined in accordance with system requirements and user preference. For example, a select query can be used to initialize slots in a CLOS instance; conversely, accessing a CLOS slot may trigger an implicit functional query.

23.1.2 Supported databases

Common SQL supports connections to various databases using the driver/client libraries for each interface-platform combination as indicated below in Table 23.1.

Common SQL may work, but is currently untested, with driver/interface/platform combinations shown as "None tested". We would be pleased to hear of your experience with these other driver/interface/platform combinations, at lisp-support@lispworks.com::

Table 23.1 Supported driver/client libraries for each interface-platform combination

<table>
<thead>
<tr>
<th>interface (module name)</th>
<th>&quot;odbc&quot;</th>
<th>&quot;oracle&quot;</th>
<th>&quot;postgresql&quot;</th>
<th>&quot;mysql&quot;</th>
<th>&quot;sqlite&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default database type</td>
<td>:odbc-driver</td>
<td>:oracle</td>
<td>:postgresql</td>
<td>:mysql</td>
<td>:sqlite</td>
</tr>
<tr>
<td>Other database type</td>
<td>:odbc</td>
<td>:oracle8</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Windows</td>
<td>Microsoft SQL Server</td>
<td>Oracle 9i(r2) and later</td>
<td>Postgres</td>
<td>MySQL</td>
<td>SQLite 3.6.12 or later</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>MySQL Postgres</td>
<td>Oracle 10g and later</td>
<td>Postgres</td>
<td>MySQL</td>
<td>SQLite 3.6.12 or later</td>
</tr>
<tr>
<td>Linux</td>
<td>MySQL Postgres</td>
<td>Oracle 9i(r2) and later</td>
<td>Postgres</td>
<td>MySQL</td>
<td>SQLite 3.6.12 or later</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>None tested</td>
<td>Not supported</td>
<td>Postgres</td>
<td>MySQL</td>
<td>SQLite 3.6.12 or later</td>
</tr>
<tr>
<td>Solaris/Intel</td>
<td>None tested</td>
<td>Oracle 10g and later</td>
<td>Postgres</td>
<td>MySQL</td>
<td>None tested</td>
</tr>
</tbody>
</table>
The keyword shown in the second and third rows is the corresponding value of the `database-type` argument to `connect`. When a client library version is shown, it is the earliest version that was tested successfully: later versions should work too, and in many cases earlier versions may work too.

**Note:** MySQL versions prior to 4.1.1 should be run in ANSI mode to work with Common SQL. That is, `mysqld` must be started with `--ansi` or the `ansi` option must appear in the `[mysqld]` section of its configuration file.

**Note:** To use PostgreSQL on any non-Microsoft Windows platform, LispWorks/Common SQL requires PostgreSQL version >= 8.x built with `--enable-thread-safety`.

## 23.2 Initialization

The initialization of Common SQL involves three stages. Firstly the SQL interface is loaded. Next, the database type (actually class) to be used is initialized. Finally, Common SQL is used to connect to a database. These stages are explained in more detail in this section.

The Lisp symbols introduced in this chapter are exported from the `sql` package. Application packages requiring convenient access to these facilities should therefore use the `sql` package.

The examples in this chapter assume that the `sql` package is visible.

### 23.2.1 Initialization steps

Three steps are required to initialize the SQL interface:

1. At load time, the SQL interface is loaded.
23.2 Initialization

2. At run time, database type(s) are initialized. This step can be merged into step 3.

3. A connection is made to a database server. All further operations use the connection.

The remainder of this section describes how you perform these steps.

1. Load the SQL interface by calling \texttt{require} with the name of a database interface.

   Currently implemented interfaces are "oracle", "mysql", "odbc", "postgresql" and "sqlite". However, not all platforms support all interfaces, see Table 23.1, page 333 for details.

   The same application can use more than one interface, and needs to call \texttt{require} to load each interface that it uses.

   Loading is done at load time. In particular, if you are building an application, loading needs to be done before calling \texttt{deliver}.

2. Initialize the database type, either when connecting or by an explicit call.

   Every connection has a database type, which defines the functionality to use when performing operations on it. Each interface defines one or more database types that can used as the database type. The database type must be initialized, which can be done either when connecting, or by explicitly calling \texttt{initialize-database-type}. Initializing a database type must be done at run time, in other words you should not save an image (by \texttt{save-image} or \texttt{deliver}) with an initialized database type.

   Initializing a database type typically means that the system finds the library that implements the client, loads and initializes it. (Actually, there may be several libraries.) It is possible to delay the initialization until making the connection, but it is useful to do the initialization explicitly first as this allows you to catch errors in the initialization and report them.

   The variable \texttt{*default-database-type*} holds the value of the default type, which is used when a database type is not supplied. The first database interface that is loaded sets \texttt{*default-database-type*} to its default database type. Therefore in a typical setup using one interface you do not need to specify the database type.
The database types currently supported are shown in Table 23.2:

<table>
<thead>
<tr>
<th>Interface</th>
<th>database-type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;oracle&quot;</td>
<td>:oracle</td>
<td>default</td>
</tr>
<tr>
<td>&quot;oracle&quot;</td>
<td>:oracle8</td>
<td>for backwards compatibility</td>
</tr>
<tr>
<td>&quot;mysql&quot;</td>
<td>:mysql</td>
<td>default</td>
</tr>
<tr>
<td>&quot;postgresql&quot;</td>
<td>:postgresql</td>
<td>default</td>
</tr>
<tr>
<td>&quot;odbc&quot;</td>
<td>:odbc-driver</td>
<td>default</td>
</tr>
<tr>
<td>&quot;odbc&quot;</td>
<td>:odbc</td>
<td>uses SQLConnect rather than SQLDriverConnect</td>
</tr>
<tr>
<td>&quot;sqlite&quot;</td>
<td>:sqlite</td>
<td>default</td>
</tr>
</tbody>
</table>

3. Connect to a database by calling `connect`.

The main argument to `connect` is a connection-spec, which is interpreted in a database type specific way. See the entry for `connect` for details. By default, `connect` uses the database type in `*default-database-type*`, but it can be specified explicitly by the keyword argument :database-type. If the database type was not initialized yet, `connect` initializes it.

The result of `connect` is an object which is referred to as "database object", but it is really a connection object representing a connection to the server. It is possible to have multiple database objects connected independently to the same database server.

The database object is used by all the other Common SQL interface functions. `connect` sets the value of `*default-database*` to the result each time it is called, so a call to a SQL interface function without specifying the database always acts on the last connected database. That allows simpler code when there is only one connection. When there is more than one connection, you need to pass the database object to the interface function via the keyword argument :database.
23.2 Initialization

When a connection is no longer required, it should be closed by calling `disconnect`.

The minimal code to initialize a connection is loading the code and connecting. For example, using only Oracle:

```lisp
(require "oracle")
(sql:connect "scott/tiger")
```

However, if you deliver an application then the `require` call needs to happen at load time (before calling `deliver`), while the `connect` call must happen at run time after the delivered application started. So your code should have two parts:

- In the script that loads the application code:
  ```lisp
  (require "oracle")
  ```
- In the code itself, at various places:
  ```lisp
  (sql:connect "scott/tiger")
  ```

To get better error handling, you may want to add a call to `initialize-database-type`, in the startup function:

```lisp
(handler-case
  (sql:initialize-database-type)
  (error (cc)
    ;; tell the user of failure to initialize Oracle
    ))
```

### 23.2.2 Database libraries

**Note:** This section applies only to Unix-like operating systems.

To use a database, LispWorks needs to load foreign libraries, which is done when initializing the database type. To find the right libraries and initialize them, there may be vendor-specific environment variable(s) that need to be set, for example `ORACLE_HOME` for Oracle. Typically one of these will point to a directory where the database code is installed. You may need to ensure that these variables are set properly when your application is used.

In order to override the default loading of database library code, you may set `*sql-libraries*`. To control messages while loading the libraries, set `*sql-loading-verbose*`. 
23.2.3 General database connection and disconnection

Database connections can be named by passing the :name argument to connect, allowing you to have more than one connection to a given database. If this is omitted, then a unique database name is constructed from connection-spec and a counter. Connection names are compared with equalp.

To find all the database connection instances, call the function connected-databases. To retrieve the name for a connection instance, call database-name, and to find a connection instance with a given name use find-database. To print status information about the existing connections, call status.

To close a connection to a database, use disconnect.

To reestablish a connection to a database, use reconnect.

23.2.3.1 Connection example

The following example assumes that the :odbc database type has been initialized as described in “Initialization steps” on page 334. It connects to two databases, scott and personnel, and then prints out the connected databases.

(setf *default-database-type* :odbc)
(connect "scott")
(connect "personnel" :database-type :odbc)
(print *connected-databases*)

23.2.4 Connecting to Oracle

For database-type :oracle, connection-spec conforms to the canonical form described for connect. The connection part is the string used to establish the connection. When connecting to a local server, it may be the SID, otherwise it is an alias recognized by the names server, or in the tnsnames.ora file.

To connect to Oracle via SQL*Net, connection-spec is of the form username/password@host where host is an Oracle hostname.

Common SQL uses the Oracle Call Interface internally where this is available. For Oracle version 8, Common SQL automatically uses the same API as in LispWorks 4.4. On some platforms, this can also be obtained by using database-type :oracle8. Note that the :oracle8 database type is restricted
because it cannot access or manipulate LOBs and all connections must use the same character set.

### 23.2.5 Connecting to ODBC

For `database-type :odbc` or `:odbc-driver`, `connection-spec` may take the canonical form described for `connect`, but an additional syntax is also allowed.

`connect` keyword arguments `:encoding`, `:signal-rollback-errors` and `:date-string-format` are all ignored.

#### 23.2.5.1 Connecting to ODBC using a string

`connection-spec` should have one of the forms:

- `username/password@dsn` The general form.
- `dsn/username/password` For backward compatibility.

The two forms of strings are distinguished by the presence (or absence) of the '@' character. In both forms, `password` can be omitted along with the preceding '/'. Also, `username` can simply be omitted.

Note that this means that "xyz" and "@xyz" are both interpreted to give the same values (`username` is null, `password` is null, `dsn` is "xyz").

#### 23.2.5.2 Connecting to ODBC using a plist

In the plist, the acceptable keywords are `:username`, `:password`, `:dsn` and `:connection`.

`:connection` is a synonym of `:dsn`.

### 23.2.6 Connecting to MySQL

For `database-type :mysql`, `connection-spec` may be in the canonical form described for `connect`, but it may also have the extensions described in this section.

In both the string and plist forms of `connection-spec` described below, any part that is omitted defaults to the MySQL default:
23 Common SQL

- **username**: anonymous user
- **password**: No password
- **dbname**: No default database
- **hostname**: localhost
- **port**: 3306 (unless using `unix-socket`).

### 23.2.6.1 Connecting to MySQL using a string

A `connection-spec` can be a string of the form:

```
username/password/dbname@hostname:port
```

where `port` is a decimal number specifying the port number to use. `port` can be omitted along with the preceding `:`.

`hostname` can be omitted. If `port` is omitted too, the `@` can be omitted as well. If `port` is supplied and `hostname` is not supplied, then both the `@` and the `:` are required, for example:

```
me/my-password/my-db@:3307
```

`hostname` may also specify a unix socket name, which must start with the character `/`.

- **dbname** may be omitted along with the preceding `/`.
- **password** may be omitted. If `dbname` is also omitted, the preceding `/` can be omitted too.
- **username** may be omitted.

### 23.2.6.2 Connecting to MySQL using a plist

A `connection-spec` can be a plist containing (some of) the keywords `:username`, `:password`, `:dbname`, `:hostname`, `:port`, `:connection`, `:unix-socket`.

Each of these keywords may be omitted.

If `:unix-socket` is specified, then none of `:hostname`, `:port` and `:connection` can be specified. If `:hostname` is specified then `:connection` must not be specified. The value supplied for `:hostname` can be a raw hostname, or a
string of the form `hostname:port`. If `:connection` is specified then it can be a string conforming to one of these patterns:

```
hostname
hostname:port
:port
unix-socket
```

Must start with '/'

That is, the value `connection` supplied in a plist `connection-spec` is interpreted just like the part of a string `connection-spec` following the `@` character.

### 23.2.6.3 Locating the MySQL client library

The MySQL interface to initialize, it must find the appropriate MySQL client library. The special variables `*mysql-library-path*`, `*mysql-library-directories*` and `*mysql-library-sub-directories*` give you control over this.

### 23.2.6.4 Special instructions for MySQL on Mac OS X

Download the 32-bit or 64-bit MySQL package to match your LispWorks image.

The downloadable packages from the MySQL web site contain only static client libraries, but LispWorks needs a dynamic library. You need to create the dynamic library, for example by using the following shell command.

To build the 32-bit dynamic library:

```
gcc -dynamiclib -fno-common \
    -o /usr/local/mysql/lib/libmysqlclient_r.dylib \
    -all_load /usr/local/mysql/lib/libmysqlclient_r.a -lz
```

To build the 64-bit dynamic library:

```
gcc -m64 -dynamiclib -fno-common \
    -o /usr/local/mysql/lib/libmysqlclient_r.dylib \
    -all_load /usr/local/mysql/lib/libmysqlclient_r.a -lz
```

This command should be executed as the root user, or some other user with write permission to the `/usr/local/mysql/lib` directory and assumes that
MySQL was installed in /usr/local/mysql, which is the location used by the prepackaged downloads.

An alternate way to create a dynamic library is to build MySQL from its source code with the --enable-shared flag.

By default, LispWorks expects to find the library either in /usr/local/mysql/lib or on the shared library path. This can be overridden by setting the special variable *mysql-library-directories*.

By default, LispWorks expects the library to be called libmysqlclient.*.dylib and it searches for a library that matches that pattern, where * is any version number. This search can be avoided by setting *mysql-library-path* to something other than the default (*-1mysqlclient*), for example, it is possible to force LispWorks to look for version 12 by evaluating

```
(setq *mysql-library-path* "libmysqlclient.12")
```

You can also set *mysql-library-path* to a full path, which avoids the need to set *mysql-library-directories*.

If the environment variable LW_MYSQL_LIBRARY is set, then its value is used instead of the value of *mysql-library-path*.

### 23.2.7 Connecting to PostgreSQL

For database-type :postgresql, connection-spec must be either a string in the format specified by the PostgreSQL libraries or a plist.

#### 23.2.7.1 Connecting to PostgreSQL using a string

If connection-spec is a string then it should be in the format specified by

http://www.postgresql.org/docs/9.3/static/libpq-connect.html#LIBPQ-CONNSTRING

For example,

```
dbname=test user=scott password=tiger host=scandium
```
23.2.7.2 Connecting to PostgreSQL using a plist

`connection-spec` can be a plist containing (some of) the keywords `:username` (or `:user`), `:password`, `:dbname`, `:hostname` (or `:host`), `:port`, `:connection`. Each of these keywords may be omitted, but if `:connection` is specified, then `:hostname` and `:port` must not be specified.

The value supplied for `:hostname` can be a raw hostname or a string of the form `hostname:port`. The value supplied for `:post` can be an integer or a string naming a service.

If `:connection` is specified then it can be a string conforming to one of these patterns:

```
  hostname
  hostname:port
```

The values should not be escaped or quoted: LispWorks will escape and quote it as needed before passing it to the PostgreSQL library.

23.2.7.3 Escaping and standard_conforming_strings

LispWorks sets the PostgreSQL session variable `standard_conforming_strings` to `on` to match the escaping that Common SQL uses. Note that this variable is only available in PostgreSQL 8.2 and later, so escaping will not work correctly in older versions of PostgreSQL.

23.2.8 Connecting to SQLite.

For `database-type :sqlite`, `connection-spec` is used to specify the filename of the SQLite database. `connection-spec` must be one of the following:

- A string. Specifies the filename as is.
- A plist containing `:dbname filename`.

  If `filename` is a string, it specifies the filename as is. Otherwise, the value of `(namestring filename)` is used as the filename.

- `:memory` This is equivalent to `":memory:"` and specifies a private, temporary in-memory database.
This is equivalent to "" and specifies a private, temporary on-disk database.

23.2.8.1 Locating the SQLite client library

The special variable sql:*sqlite-library-path* contains the FLI shared library name for SQLite. It defaults to "-lsqlite3" on non-Windows platforms, which should work if SQLite is installed. On Windows, it defaults to "sqlite3.dll", which requires that DLL to be on the path. Note that 64-bit and 32-bit LispWorks require different DLL files. The Common SQL SQLite interface assumes that the library is compiled with standard options and that SQLite is not configured in an unusual way. Most importantly, if the threading mode is single-thread (either because the library is compiled as single-thread, or because sqlite3_config set it to single-thread), then the Common SQL SQLite interface is not thread-safe anymore. This situation is quite unlikely to happen.

23.2.8.2 SQLite string encoding

By default, the connection is opened as a UTF-8 connection (using the C function sqlite3_open_v2). The :encoding argument to connect can have the value :default, :unicode or :utf-8 which all use the default (that is, have no effect), and :utf-16 or :utf-16-native, which opens the connection using UTF-16 in the native byte order (using the C function sqlite3_open16). It is not obvious in what circumstances UTF-16 is better and it is made available only because the underlying library supports it. When opening as UTF-16, the keywords :open-mode, :threading-mode, :uri and :vfs are ignored.

23.2.8.3 SQLite connection keywords

The sqlite-keywords keyword argument to connect allows you to specify several parameters controlling the behavior of the connection. sqlite-keywords is a property list, providing values for the SQLite-specific keywords :open-mode, :threading-mode, :uri, :cache-mode, :vfs or :uniform-type-per-column. These keywords affect the connection as follows.

By default, the connection is opened with opening modes create and read-write, which means that the file is created if it does not exist, and the database
can be modified. The SQLite-specific keyword `:open-mode` in `sqlite-keywords` can be used to change this. The value `:readonly` specifies that the file must exist (so `connect` fails if it does not exist) and is opened for reading only (so it is not possible to modify it). The value `:readwrite` means that the file must exist and the database can be modified.

By default, the threading mode of the connection is "serialized" (so it is thread-safe). You can change this by the SQLite-specific keyword `:threading-mode` in `sqlite-keywords`, which can take the values `:multi-thread` or `:serialized`. When the threading mode is `:multi-thread` (rather than `:serialized`), it is not actually thread-safe, and you can access it only on one thread at a time (but it can be accessed from different threads over time). The term "multi-thread" means that you can access different connections at the same time on different threads. `:multi-thread` is probably more efficient, but we do not know by how much.

By default, `connection-spec` can be a URI filename, which is a string starting with "file:" (see https://www.sqlite.org/uri.html "URI Filenames In SQLite" for details). Whether this is allowed is controlled by the SQLite-specific keyword `:uri` in `sqlite-keywords`, which defaults to `t`, and can be switched off by passing `:uri nil`.

By default, the connection cache mode is the system default. The SQLite-specific keyword `:cache-mode` in `sqlite-keywords` can be used to change this to either `:private` or `:shared`. See https://www.sqlite.org/sharedcache.html "SQLite Shared-cache mode" for details. `:Shared` mode probably improves performance if you connect multiple times to the same file.

The SQLite-specific keyword `:vfs` in `sqlite-keywords` can be used to specify the name of the VFS (Virtual File System) that is used. You need to be an expert on SQLite to know when this is useful.

The SQLite-specific keyword `:uniform-type-per-column` in `sqlite-keywords` affects the default type for fields in the results of queries. See “Tables containing a uniform type per column” on page 394.

See “Using SQLite” on page 392 for other SQLite-specific features.
23.3 Functional interface

The functional interface provides a full set of Data Manipulation and Data Definition functions. The interface provides a SQL-compatible means of querying and updating the database from Lisp. In particular, the values returned from the database are Lisp values — thus smoothly integrating user applications with database transactions. An embedded syntax is provided for dynamically constructing sophisticated queries through `select`. Iteration is also provided via a mapping function and an extension to the `loop` macro. If necessary, the basic functions `query` and `execute-command` can be called with SQL statements expressed as strings. It is also possible to update or query the data dictionary.

23.3.1 Functional Data Manipulation Language (FDML)

The functions available for Data Manipulation and Data Definition are described below.

23.3.1.1 Querying

The function `select` returns data from a database matching the constraints specified. The data is returned, by default, as a list of records in which each record is represented as a list of attribute values.

Database identifiers used in `select` are conveniently specified using the symbolic SQL `[ ]` syntax. This syntax is enabled by calling `enable-sql-reader-syntax`.

The square bracket syntax assumes that sql symbols are visible. Therefore when using the `[ ]` syntax, ensure that the current package either is `sql`, or is a package which has the `sql` package on its package-use-list.

For a description of the symbolic SQL syntax see Section 23.5 on page 358. For example, the following is a potential query and its result:

```
(select [person_id] [person surname] :from [person]) =>
((111 "Brown") (222 "Jones") (333 "Smith"))
("PERSON_ID" "SURNAME")
```
In this example, [person_id], [person surname] and [person] are database-identifiers and evaluate to literal SQL. The result is a list of lists of attribute values. Conversely, consider

\[
\text{(select [surname] :from [person] :flatp t)}
\]

\[
= \text{["Brown" "Jones" "Smith" ]
}("SURNAME")
\]

In this case the result is a simple list of surname values because of the use of the flatp keyword. The flatp keyword only works when there is one column of data to return.

In this example we use * to match all fields in the table, and then we use the result-types keyword to specify the types to return:

\[
\text{(select [*] :from [person])}
\]

\[
= \text{[(2 111 "Brown") (3 222 "Jones") (4 333 "Smith") ]
("ID" "Person_ID" "Surname")
\]

\[
\text{(select [*] :from [person] :result-types '(:integer :string :string))}
\]

\[
= \text{[(2 "111" "Brown") (3 "222" "Jones") (4 "333" "Smith") ]
("ID" "Person_ID" "Surname")
\]

If you want to affect the result type for a specified field, use a type-modified database identifier. As an example:

\[
\text{(sql:select [Person_ID :string][Surname] :from [person])}
\]

\[
= \text{["111" "Brown" ]
("PERSON_ID" "SURNAME")
\]

With database-type :mysql, further control over the values returned from queries is possible as described in “Types of values returned from queries” on page 377.

In this final example the :where keyword is used to specify a condition for returning selected values from the database.

\[
\]

\[
= \text{["Jones"]
("SURNAME")
\] }
To output the results of a query in a more easily readable tabulated way, use the function `print-query`. For example the following call prints two even columns of names and salaries:

```lisp
(print-query [select [surname] [income] :from [employee]]
             :titles '("NAME" "SALARY")
NAME   SALARY
Brown  22000
Jones  45000
Smith  35000
```

### 23.3.1.2 Modification

Modifications to the database can be done using the following functions; `insert-records`, `delete-records` and `update-records`. The functions `commit`, `rollback` and the macro `with-transaction` are used to control transactions. Although `commit` or `rollback` may be used in isolation it is advisable to do any updates inside a `with-transaction` form instead. This provides consistency across different database transaction models. For example, some database systems do not provide an explicit “start-transaction” command while others do. `with-transaction` allows user code to ignore database-specific transaction models.

The function `insert-records` creates records in a specified table. The values can be either specified directly with the argument `values` or in the argument `av-pairs`, or they can be the result of a query specified in the `query` argument. The attributes can be specified with the argument `attributes` or in the argument `av-pairs`.

If `attributes` is supplied then `values` must be a corresponding list of values for each of the listed attribute names. For example, both:

```lisp
(insert-records :into [person]
                :attributes '(person_id income surname occupation)
                :values '(115 11000 "Johnson" "plumber")
```

and:

```lisp
(insert-records :into [person]
                :attributes '(person_id income surname occupation)
                :values '(115 11000 "Johnson" "plumber")
```
23.3 Functional interface

(insert-records :into [person]
  :av-pairs '((person_id 115)
    (income 11000)
    (surname "Johnson")
    (occupation "plumber"))
)

are equivalent to the following SQL:

```
INSERT INTO PERSON
  (PERSON_ID, INCOME, SURNAME, OCCUPATION)
VALUES (115, 11000, 'Johnson', 'plumber')
```

If query is provided, then neither values nor av-pairs should be. In this case the attribute names in the query expression must also exist in the insertion table. For example:

```
(insert-records :into [person]
  :query [select [id] [firstname] [surname]
    :from [manager]]
  :attributes '(person_id firstname surname))
```

To delete or alter those records in a table which match some condition, use delete-records or update-records.

23.3.1.3 Caching of table queries

Operations which add or modify records sometimes need to perform an internal query to obtain type information for the relevant attributes. In principle it is possible for the database schema to change between update operations, and hence this query is run for each update operation. This can be a significant overhead.

For tables which are guaranteed to have a constant schema, you can optimize performance by adding a cache of these internal query results, using the function cache-table-queries. This can also be used to reset the cache if the table schema is actually altered. To control the default caching behavior throughout every database connection, you can set the variable *
```
23.3.1.4 Transaction handling

A transaction in SQL is defined as starting from the connect, or from a commit, rollback or data-dictionary update and lasting until a commit, rollback, data-dictionary update or a disconnect command.

The macro with-transaction executes a body of code and then does a commit, unless the body failed in which case it does a rollback. Using this macro allows your code to cope with the fact that transactions may be handled differently in the different vendor implementations. Any differences are transparent if the update is done within a with-transaction form.

Note: Common SQL opens an ODBC database in manual commit mode, so that with-transaction and rollback take effect.

Applications should perform all database update operations in a with-transaction form (or follow them with commit or rollback) in order to safely commit or discard their changes. This applies to operations that modify either the data or the schema.

The following example shows a series of updates to an employee table within a transaction. This example would commit the changes to the database on exit from with-transaction. This example inserts a new record into the emp table, then changes those employees whose department number is 40 to 50 and finally removes those employees whose salary is more than 300,000.

```
(connect "personnel")
(with-transaction
  (insert-records :into [emp]
      :attributes '(empno ename job deptno)
      :values '(7100 "ANDERSON" "SALESMAN" 30))
  (update-records [emp]
      :attributes [deptno]
      :values 50
      :where [= [deptno] 40])
  (delete-records :from [emp]
      :where [> [sal] 300000]))
)
```

To commit or roll back all changes made since the last commit, use the functions commit or rollback.
23.3 Functional interface

23.3.1.5 Iteration

Common SQL has three iteration constructs: a do loop, a mapping function, and an extension to the Common Lisp loop macro.

The macros do-query and simple-do-query repeatedly execute a piece of code within the scope of variables bound to the attributes of each record resulting from a query.

The function map-query maps a function across the results of a query and returns its result in a sequence of a specified type, like the Common Lisp map function.

Common SQL provides an extension to the ANSI Common Lisp macro loop which is a clause for iterating over query results. The syntax of the clause is:

```
{for|as} var [type-spec] being
{the|each}{tuples|tuple}
{in|of} query-expression
```

query-expression can be a string, a sql-expression-object (a result of the "[...]" syntax) or a prepared-statement.

The more general word tuple is used so that it can also be applied to the object-oriented case. In the functional case, tuple is synonymous with record.

Each iteration of the loop assigns the next record of the table to the variable var. The record is represented in Lisp as a list. Destructuring can be used in var to bind variables to specific attributes of the records resulting from query-expression. In conjunction with the panoply of existing clauses available from the loop macro, the new iteration clause provides an integrated report generation facility.

Suppose the name of everyone in an employee table is required. This simple query is shown below using the different iteration method. The function map-query requires flatp to be specified; otherwise each name would be wrapped in a list.

```
(do-query ((name) [select [ename] :from [emp]])
  (print name))
```
The following extended loop example binds, on each record returned as a result of the query, name and salary, accumulates the salary, and for salaries greater than 2750 increments a count, and prints the details. Finally, the average salary is printed.

```
(loop for (name salary) being each record in
  [select [ename] [sal] :from [emp]]
  initially (format t "~&~20A~10D" 'name 'salary)
  when (and salary (> salary 2750))
    count salary into salaries
    and sum salary into total
    and do (format t "~&~20A-10D" name salary)
  else
    do (format t "&-2A&v Salary: ~10D" (/ total salaries)))
```

### 23.3.1.6 Specifying SQL directly

Sometimes it is necessary to execute vendor-specific SQL statements and queries. For these occasions Common SQL provides the functions **query** and **execute-command**. They can also be used when the exact SQL is known in advance and thus the square bracket syntax is not needed. The query expression can be a string, a sql-expression-object (a result of the "[...]" syntax) or a prepared-statement.

The function **query** runs a SQL query on a database and returns a list of values like select (see “Querying” on page 346). It also returns a list of the field names selected.

**execute-command** is the basic function which executes any SQL statement other than a query. It can run a stored procedure, as described in **execute-command**, page 1567.
23.3 Functional interface

23.3.1.7 Building vendor-specific SQL

Common SQL does not provide a general interface to vendor-specific syntax. There are two approaches you can take with SQL such as this:

```sql
SELECT B.PARTY_CODE_ALIAS, A.VALUE FROM CODES A, CODE_ALIASES B
WHERE A.DOMAIN=B.CODE_DOMAIN(+) AND A.VALUE=B.CODE_VALUE(+)
AND B.PARTY_ID(+)=<party_id>
```

1. Construct the string as above and then call `query` as described in “Specifying SQL directly” on page 352.

2. Use `sql-expression` to construct the vendor-specific pieces of the SQL. The above expression can be written like this:

```sql
(sql:select [b party_code_alias] [a value]
:from '((codes a) (codes_aliases b))
:where [and [= [a domain]
 (sql:sql-expression
 :string "B.CODE_DOMAIN(+)"
 ] = (sql:sql-expression
 :string "B.PARTY_ID(+)")) PARTY-ID]))
```

23.3.2 Functional Data Definition Language (FDDL)

Functions in the FDDL may be used to change or query the structure of the database.

23.3.2.1 Querying the schema

The functions `list-tables`, `list-attributes`, `attribute-type` and `list-attribute-types` return information about the structure of a database.

23.3.2.2 FDDL Querying example

This example shows you how to query the type of the `ename` attribute of the `emp` table.

```sql
(attribute-type [ename] [emp]) -> :char
```
23.3.2.3 Modification

You may create or drop (delete) tables using the functions `create-table` and `drop-table`.

Create or drop indexes using the functions `create-index` and `drop-index`.

To create or drop a view (that is, a derived table based on a query) use the functions `create-view` and `drop-view`.

23.4 Object oriented interface

This section describes the object-oriented interface to SQL databases using specialized CLOS classes. These classes have `standard-db-object` as one of their superclasses and have a common metaclass which provides the specialized behavior for mapping subclasses of `standard-db-object` onto records in the database. A class of this kind is created using `def-view-class`.

23.4.1 Object oriented/relational model

In the simple case, a class maps onto a database table, an instance of the class maps onto a record in the table, and a slot in the class maps onto an attribute in the table.

In general, however, a class maps onto a database view, an instance of the class maps onto a collection of records in the view, and a slot in the class is either:

- A base slot that maps onto an attribute in the view
- A join slot that points to a list of other view-class instances

If an instance maps onto more than one record in the view then for each record, all the key attributes from each table in the view are the same.

23.4.1.1 Inheritance for View Classes

It is not possible to inherit from a class that was defined by `def-view-class`. All of the slots need to be in the same class (and hence also in the same SQL table).
23.4 Object oriented interface

23.4.2 Object-Oriented Data Definition Language (OODDL)

The OODDL lets you define a mapping between the relational and object-oriented worlds to be defined. Through the mapping a CLOS object can effectively denote a collection of records in a database view, and can contain pointers to other view-based CLOS objects. The CLOS object makes explicit an object implicitly described by the flat relational values.

The mapping is defined using the macro `def-view-class`. This extends the syntax of `defclass` to allow special base slots to be mapped onto the attributes of database views (presently single tables). When you submit a `select` query that names a View Class (that is, a class defined by `def-view-class`), then the corresponding database view is queried, and the slots in the resulting instances are filled with attribute values from the database.

It is also possible to create join slots and virtual (ordinary) slots.

All the special slots are distinguished by a modified set of class and slot options. The special slots and their options are described in more detail under `def-view-class` in the LispWorks Reference Manual.

Note: `def-view-class` defines a Lisp view of an underlying database table. It is a similar concept to that of SQL VIEWs, but does not interact with them.

You can create a table based on a View Class using the function `create-view-from-class` and delete it using the function `drop-view-from-class`.

23.4.2.1 Example View Class definition

The following example shows a View Class corresponding to the traditional employees table, with the employee’s department given by a join with the departments table. See `def-view-class`, page 1551 for a description of the slot options.
(def-view-class employee (standard-db-object)
  ((employee-number :db-kind :key
    :column empno
    :type integer)
  (employee-name :db-kind :base
    :column ename
    :type (string 20)
    :accessor employee-name)
  (employee-department :db-kind :base
    :column deptno
    :type integer
    :accessor employee-department)
  (employee-job :db-kind :base
    :column job
    :type (string 9))
  (employee-manager :db-kind :base
    :column mgr
    :type integer)
  (employee-location :db-kind :join
    :db-info (:
      :join-class department
      :retrieval :deferred
      :set nil
      :home-key employee-department
      :foreign-key department-number
      :target-slot department-loc)
    :accessor employee-location))
 (:base-table emp))

The def-view-class macro allows elements or lists of elements to follow :home-key and :foreign-key. The elements can be symbols, nil, strings, integers or floats.

This syntax means that an object from the join class is only included in the join slot if the values from home-key are equal to the values in foreign-key, in order. These values are calculated as follows:

- If the element in the list is a symbol it is taken to be a slot name and the value of the slot is used
- Otherwise the element is taken to be the value

Note that some database vendors may have short maximum identifier lengths. The CLOS interface uses constructed alias names for tables in its SQL queries, and long table names or long class names may cause the constructed aliases to exceed the maximum identifier length for a particular vendor.
23.4 Object oriented interface

23.4.3 Object-Oriented Data Manipulation Language (OODML)

The OODML is designed to be powerful and expressive, while remaining familiar to users of the FDML. To achieve this aim, some of the functions and macros in the SQL interface have been overloaded — particularly the select function and the iteration constructs.

The function select is common across the both the functional and object-oriented SQL interfaces. If its first argument, selections, refers to a View Class by supplying its symbolic name then the select operation becomes object-oriented and it returns a list of instances instead of a list of attributes.

A subsequent equivalent select call will return the same (eq) instances. The :refresh argument can be used to ensure that existing instances get updated with any changed data. If such an update requires action by your application, then add methods on the generic function instance-refreshed.

In a View Class select call, the symbol slot-value is a valid SQL operator for use within the :where argument.

To find the View Classes for a particular database, use the function list-classes.

To manipulate data via a View Class, that is to modify the records corresponding to instances of the View Class, using the generic functions update-records-from-instance, and update-record-from-slot.

To delete records corresponding to instances of the View Class, use the generic function delete-instance-records.

To update existing instances of a View Class when data is known to have changed, use the generic functions update-slot-from-record and update-instance-from-records.

23.4.3.1 Examples

[select 'employee]
  -> #<SQL-OBJECT-QUERY (EMPLOYEE)>
23.4.3.2 Iteration

The object-oriented SQL interface has the same three iteration constructs as the functional interface (see Section 23.3.1.5 on page 351): a do-loop, a mapping function, and an extension to the Common Lisp loop macro. However, in this case, the iteration focus is not a tuple of attributes (that is, a record), but a tuple of instances. For example:

\[
\text{(loop for \(j\) \(c\) being the tuples in}
\]
\[
\text{[select \'person \'organization}
\]
\[
\text{:where \([= \text{[slot-value \'person \'surname\] "Jones\]}
\]
\[
\text{do \{format t \"\-A \-A \-\%\"
\]
\[
\text{\[(slot-value j \'forename\]
\]
\[
\text{\(\text{[slot-value c \'short-name\]}}\})}
\]

Note: Instances may denote many database records, and hence the effective iteration focus in this case is a tuple of sets of tuples of attributes.

23.4.3.3 Garbage collection of view instances

View instance objects are not released for garbage collection (GC) until the connection is closed. This is because they are referenced by the CLOS object representing the database connection. This is to ensure that they can reliably be compared by \text{eq}.

23.5 Symbolic SQL syntax

Common SQL supports a symbolic query syntax across both the functional and object-oriented interface layers. It allows SQL and Common Lisp expressions to be mixed together — with as much processing as possible done at compile-time. Symbolic SQL expressions are read as square-bracketed lists to
distinguish them from Lisp expressions. However, each can be nested within the other to achieve the desired result.

By default, this reader syntax is turned off. To turn it on see Section 23.5.3 on page 370.

23.5.1 The “[...]” Syntax

The square bracket syntax for the SQL interface is heavily overloaded to provide the most intuitive behavior in all situations. There are three uses of square brackets:

1. To enclose a database identifier
2. To construct a SQL string representing a symbolic expression
3. To enclose an SQL expression directly

Each of these uses is demonstrated below.

23.5.1.1 Enclosing database identifiers

Database identifiers are specified in the "[...]" syntax using the following rules:

There must be one, two or three Lisp forms inside the square brackets. The first form must be a symbol, string or a recursive database identifier (that is, another square brackets expression). The second form, if present, must be a symbol or a string. The third form, if present, must be a keyword.

The case with a single form that is a string is special, and is interpreted as a direct SQL expression rather than an identifier (see “Enclosing a SQL expression directly” on page 366 below).

When a string or a symbol is used to specify all or part of the identifier and the string (or name of the symbol) cannot be used as an identifier (because it contains special characters or matches a SQL reserved word), then it is wrapped with double quotes in the resulting SQL.

If there is more than one form inside the square brackets, and the first form is a symbol that is recognized as a SQL operator or a pseudo-operator, then the expression is interpreted as an operation rather than as an identifier (see the following sections).
The first form is always interpreted as specifying a string that is part or all of the identifier. For a symbol, it is the symbol name and for a recursive identifier it is the string that would be generated for this identifier. In the examples below, the text following the => (and optionally up to the semicolon) shows what is generated for the resulting SQL.

If there is only one form, it specifies the full name of the identifier. For example:

- [foo] => FOO
- ["foo"] => foo
- [[foo]] => FOO
- ["W%{(jj"] => W%{(jj ; single form string not quoted.

If the second form is a string and the first form is not a string, then the first form specifies the name of the identifier and the second form specifies an alias. In this case there must not be a third form. The alias identifier is useful for giving tables aliases in the from part of the SQL select statement:

- [foo "AA"] => FOO AA
- [[foo aa] "bb"] => FOO.AA bb ; first form is recursive.

If there is a third form, or the second form is not keyword, or the first form is a string, then the second form specifies an identifier qualified by the first form, that is they are combined with a period in the middle:

- [foo aa] => FOO.AA
- [foo aa :integer] => FOO.AA ; with type :integer (below).
- ["foo" "AA"] => foo.AA ; compare to [foo "AA"] above.

If there are only two forms and the second form is a keyword, or there are three forms, then the second form (in the two form case) or the third form (in the three form case) specifies a type associated with the identifier. The type
23.5 Symbolic SQL syntax

does not affect the SQL statement that the database sees. It is used when the identifier is part of the selection list, to tell Common SQL what type the value should be. Such identifiers should appear only in the selection list of queries.

```
[ColumnName :integer]
=> COLUMNNAME ; type :integer.
```

```
[[TableName ColumnName] :string]
=> TABLENAME.COLUMNNAME ; type :string.
```

```
[TableName ColumnName :string]
=> TABLENAME.COLUMNNAME ; type :string
(same as previous).
```

Inside select (which is recognized as a SQL operator):

```
[select [id :integer] [name :string] :from [TableName]]
=> SELECT ID, NAME FROM TABLENAME ; interpret
ID as an integer and NAME as a string.
```

Notes:

- You can specify both an alias and a type by specifying the identifier recursively in the first form:

```
[[TableName ColumnName] "MyAlias" :string]
=> TABLENAME.COLUMNNAME MyAlias ; type :string
```

- Recursion through the first form also allows you to add qualifiers as needed:

```
[[[[CatalogName SchemaName TableName] ColumnName] TableName] ColumnName] "MyAlias" :string]
=> CATALOGNAME.SCHEMANAME.TABLENAME.COLUMNNAME MyAlias ; type :string.
```

- Because a string as single form is not quoted, it allows you to insert any SQL directly. For example, in the first expression below the string which contains illegal characters is quoted, but in the second example the
string appears as a single form in the recursive identifier, so is not quoted:

```sql
["W%((jj" aa]
```

=> "W%((jj".AA ; string is quoted.

```sql
[["W%((jj"] . aa]
```

=> W%((jj.AA ; string not quoted because it is a single form.

- Evaluating an expression in "[...]" syntax returns an object of type `sql-expression-object`.

### 23.5.1.2 Specifying the type of retrieved values.

When you use a keyword to specify the type of an expression as described in "Enclosing database identifiers" on page 359, you are telling common SQL that the values retrieved for this expression should be of a specific type. For example, if you call:

```sql
(sql:select [name :string] :from [TableNane])
```

then the :string keyword tells common SQL that the values for `name` should be strings.

There are four keywords that are supported by all common SQL backends: :string, :integer, :double-float and :single-float. For each of these keywords, the values are mapped to the matching Common Lisp type. If this is not possible, the value is returned as `nil`.

Note that if you specify a keyword that is incompatible with the type in the database column then either an error is signaled or all returned values will be `nil`.

The keyword :int is accepted as an alias for :integer.

The keyword :binary is supported by most of backends (except Microsoft Access and PostgreSQL). The value that is returned for :binary is an array with element type `(unsigned-byte 8)`. On Oracle, :binary can be used only for columns of binary type, so it is only useful when you want to retrieve the
23.5  *Symbolic SQL syntax*

contents of a BLOB directly, because for plain RAW columns it is the default anyway. Other backends allow you to retrieve at least strings as binary values. Other keywords are supported by some of the backends, and are documented in the backend specific sections.

**23.5.1.3 Symbolic expression of SQL operators**

When the first form in the square brackets is a symbol that is one of the SQL operators listed below, the expression is interpreted as an operation. For example:

\[
[\text{any } '(3 \ 4)] \rightarrow \text{#<SQL-VALUE-EXP } "(ANY (3,4))"\]

Similarly with two argument operators:

\[
[> [\text{baz}] \ [\text{beep}]] \\
\rightarrow \text{#<SQL-RELATIONAL-EXP } "(BAZ > BEEP)"\]

The `select` statement itself may be prepared for later query execution using the `[ ]` syntax. For example:

\[
[\text{select } [\text{person_id}] \ [\text{surname}] :\text{from } [\text{person}]]
\]

This form results in a SQL expression, which could be bound to a Lisp variable and later given to `query` to execute. For example:

\[
[\text{select } [\text{foo}] \ [\text{bar} *] \ \\
:\text{from } '((\text{baz}) \ [\text{bar}]) \ \\
:\text{where } [\text{or } [= [\text{foo}] 3] \\
\rightarrow [> [\text{baz.quux} 10]]]
\rightarrow \text{#<SQL-QUERY } "(SELECT FOO,BAR.* FROM BAZ,BAR \\
WHERE ((FOO = 3) \ \\
OR (BAZ.QUUX > 10)))"\]

Strings can be inserted in place of database identifiers within a `select`:
[select [foo bar] [baz]
 :from '([foo] [quux])
 :where [or [>] [baz] 3]
    [like [foo bar] "SU%"]]
 ->
 #<SQL-QUERY:
 "(SELECT FOO.BAR,BAZ
 FROM FOO,QUUX
 WHERE ((BAZ > 3)
 OR (FOO.BAR LIKE 'SU%')))">

Any non-constant included gets filled in at runtime, for example:

[>] [foo] x]

when macroexpanded reads as

(SQL-> #<SQL-IDENT "FOO"> X)

which constructs the actual SQL string at runtime.

Any arguments to a SQL operator that are Lisp constants are translated to the
matching SQL construct at compile-time, for example:

"foo" -> "'foo'"
3 -> "3"
'("this" 5 "that") -> "('this', 5, 'that')"

xyz -> "XYZ"

SQL operators which are supported are null, exists, *, +, /, -, like, substr,
and, or, not, in, all, any, some, |, =, <, >, >=, <=, <>, order-by, count, max,
min, avg, sum, minus, nvl, distinct, except, intersect, union, slot-
value, between and userenv. There are also pseudo operators for calling
database functions (see “Calling database functions” on page 365).

The general syntax is: [<operator> <operand> ...], for instance:

(sql:select [count [*]] :from [empl])

The operand can itself be a SQL expression, as in the following example:
23.5 Symbolic SQL syntax

```lisp
(sql:create-table [company]
  '(([name] (varchar 20) not-null)))

(loop for company in '("LispWorks Ltd" "Harlequin" "Oracle" "Rover" "Microsoft")
do
  (sql:insert-records :into [company]
    :av-pairs `(([name] ,company))))

(sql:create-table [person]
  '(([surname] (varchar 20) not-null)
    ([firstname] (varchar 20) not-null)))

(loop for person in '((("Joe" "Bloggs")
  ("Fred" "Smith")
  ("Rover" "the Dog")
  ("Fido" "the Dog"))
do (sql:insert-records :into [person]
  :av-pairs
    `(([firstname] ,(car person))
      ([surname] ,(second person)))))

(sql:select [name]
  :from [company]
  :where [= [name]
    [any [select [surname]
      :from [person]]]])

(sql:select [surname]
  :from [person]
  :set-operation [union [select [firstname]
    :from [person]]]))

23.5.1.4 Calling database functions

An arbitrary function can be included in the SQL using the pseudo operator sql-function. The first argument is the function name and the rest are its arguments, for example:

```lisp
(select [sql-function "COS" [age]] :from [EMPLOYEES])
```
(insert-records
  :into [atable]
  :attributes '(a b)
  :values
  (list 1 [sql-function "TO_DATE" "02/06/99" "mm/DD/RR"]))

Also you can call SQL infix operators using the pseudo operators sql-bool-ean-operator and sql-operator.

### 23.5.1.5 Enclosing a SQL expression directly

An SQL expression can simply be enclosed directly in the square bracket syntax, as shown below.

Creating a full query (which can be used as argument to query):

```
["SELECT FOO, BAR FROM BAZ"]
-> #<SQL "SELECT FOO, BAR FROM BAZ">
```

Using an non-portable function condition in :where:

```
(sql:select [*] :from ["aTable"]
  :where ["non_portable_function() > 89"])
```

### 23.5.1.6 SQL string literals

SQL string literals can be used as arguments to operators, for example with a constant Lisp string:

```
[= [name] "John"]
```

or with a Lisp expression that evaluates to string:

```
(defun find-person-age (name)
  (car (select [age] from [table]
      :where [= [name] name])))
```

where the argument name is a string.

However, Microsoft SQL Server (which can be used via ODBC) requires the N syntax for string literal that are not entirely ASCII, or contain characters that are not recognized by the server code page. (The N syntax prefixes the string literal by the character N, for example N'Greek', rather than 'Greek'.)

Although this syntax is part of the SQL standard, not all SQL backends accept it (in particular, SQLite and Microsoft Access (via ODBC) do not). Thus the
decision whether to use the N syntax needs to be made at run time and requires the SQL backend (which is represented by the database object that connect returns). By default, the symbolic SQL syntax does not use the N syntax, but the special pseudo-operator string can be used to override this. string takes a required argument, which must be a string, and an optional argument, a database (which defaults to *default-database*), and produces the appropriate syntax for that database. The example above can be written using string like this:

```lisp
(defun find-person-age (name)
  (car (select [age] from [table]
             :where [= [name] [string name]])))
```

The same database must be used for the string pseudo-operator and the function/macro that uses the resulting expression. In the example above, the function is select and the database not specified at all, so both string and select will use *default-database*. This restriction means that the string pseudo-operator cannot be used to generate a pre-existing expression, which is otherwise possible with the symbolic SQL syntax. For example, your code might contain:

```lisp
(defun find-person-age (name)
  (car (select [age] from [table]
             :where [= [name] [string name]])))
```

The same database must be used for the string pseudo-operator and the function/macro that uses the resulting expression. In the example above, the function is select and the database not specified at all, so both string and select will use *default-database*. This restriction means that the string pseudo-operator cannot be used to generate a pre-existing expression, which is otherwise possible with the symbolic SQL syntax. For example, your code might contain:

```lisp
(defvar *match-name-starting-with-cf* [like [name] "CF"])
```

which defines *match-name-starting-with-cf* at load time, and then use it elsewhere:

```lisp
(defun some-function (arg1 ..)
  ..
  (select [*] :from [table]
            :where *match-name-starting-with-cf*)
  ..
)
```

But if you use [string "CF"] in the defvar, it will try to use the database at load time, which is normally before the database is connected.

You can perform approximately what the string pseudo-operator does by using string-prefix-with-n-if-needed:

```lisp
(let ((maybe-qualified
        (string-prefix-with-n-if-needed name))
      (car (select [age] from [table]
                :where [= [name] maybe-qualified]))))
```
Another option is to set the variable `*use-n-syntax-for-non-ascii-strings*` to `t` at compile time, which causes all string literals that are not entirely ASCII to be produced with N syntax. That would generate code that will work with almost all SQL backends, but not with SQLite or Microsoft Access (which do not support the N syntax). The advantage is that, if you have a large number of string literals, then you do not have to change them all: you just need to recompile your code with `*use-n-syntax-for-non-ascii-strings*` set to `t`.

### 23.5.2 Programmatic interface

In some cases it is necessary to build SQL-expressions dynamically under program control.

The function `sql-operation` returns the SQL expression for an operator applied to its arguments. It also supports building SQL expressions which contain arbitrary SQL functions using the pseudo operators `sql-function`, `sql-operator` and `sql-boolean-operator`. For examples see `sql-operation`, page 1660.

The function `sql-expression` makes a SQL expression from the given keywords. This is equivalent to the first and third uses of the `{` syntax as discussed in Section 23.5.1 on page 359.

The function `sql-operator` returns the Lisp symbol for a SQL operator.

The function `sql` makes SQL out of the arguments supplied. Each argument to `sql` is turned into SQL and then the `args` are concatenated with a single space between each pair. A Lisp string maps to the same characters enclosed between single quotes (this corresponds to a SQL string constant). `nil` maps to "NULL", that is, a SQL null value. Symbols and numbers map to strings. A list maps to a parenthesised, comma-separated expression. A vector maps to a comma-separated expression, which allows the easy generation of SQL lists that require no parentheses such as table lists in select statements.

The rules for the conversion are fully specified in `sql`, page 1642.
23.5 Symbolic SQL syntax

23.5.2.1 Examples

The following example function, taken from the object-oriented SQL interface layer, makes a SQL query fragment that finds the records corresponding a CLOS object (using the slots as attributes), when built into the where-clause of an updating form.

```lisp
(let* ((class (class-of object))
       (key-slots (db-class-keyfields class)))
  (loop
    for key in key-slots
    for slot-name = (slot-definition-name key)
    for slot-type = (db-slot-definition-type key)
    collect
    [= (make-field-name class key)
       (lisp-to-sql-format
        (slot-value object slot-name)
        (if (listp slot-type)
            (car slot-type)
            slot-type))]
    into cols
  finally (apply (sql-operator 'and) cols)))
->
#<SQL-RELATIONAL-EXP "(EMP.EMPNO = 7369">"
```

Here is another example that produces a SQL select statement:

```lisp
(sql-operation 'select
  (sql-expression :table 'foo
                   :attribute 'bar)
  (sql-expression :attribute 'baz)
  :from (list
         (sql-expression :table 'foo)
         (sql-expression :table 'quux))
  :where (sql-operation 'or
            (sql-operation '>'
             (sql-expression :attribute 'baz)
             3)
            (sql-operation 'like
             (sql-expression :table 'foo
                             :attribute 'bar)
             "SU%"))
->
#<SQL-QUERY "SELECT FOO.BAR,BAZ FROM FOO,QUUX
WHERE ((BAZ > 3) OR (FOO.BAR LIKE 'SU%'))">"
```

369
23.5.3 Utilities

The function `enable-sql-reader-syntax` switches square bracket syntax on and sets the state so that `restore-sql-reader-syntax-state` restores the syntax again if it is subsequently disabled. The function `disable-sql-reader-syntax` switches square bracket syntax off and sets the state so that `restore-sql-reader-syntax-state` disables the syntax again if it is subsequently enabled.

The functions `locally-enable-sql-reader-syntax` and `locally-disable-sql-reader-syntax` switch square bracket syntax on and off, but do not change the state restored by `restore-sql-reader-syntax-state`. The intended use of these is in a file:

```
#. (locally-enable-sql-reader-syntax)
<code using [..]> #. (restore-sql-reader-syntax-state)
```

23.6 Working with date fields

This section describes particular issues around using datetime database fields via Common SQL. Note: SQLite does not support date fields at all.

See also “Types of values returned from queries” on page 377 for information specifically about returning datetime values from MySQL.

23.6.1 Testing date values

Compare DATE values by formatting the date as a string in a date format that the database can parse. For example:

```
(sql:select * :from [Table] :where [= [Date] "2005-12-25"])
```

Note that it is not possible to lookup date values in the database using numeric values. This is because:

1. Common SQL cannot know that the field will be a date field until the results are returned, and
2. the database probably does not know about Common Lisp universal time.
To convert between universal time and standard SQL DATE or TIMESTAMP string, you can use the functions `encode-db-standard-date`, `encode-db-standard-timestamp`, `decode-to-db-standard-date` and `decode-to-db-standard-timestamp`. Note that the database may have non-standard date format, in which case you will need to either format the string yourself, or on Oracle tell the database to use the standard format by passing `date-string-format` to `connect`.

### 23.6.2 DATE returned as universal time

By default Common SQL converts DATE values to Common Lisp universal times. Therefore code like this returns Common Lisp universal times (that is, integers) where `MyDate` is a DATE field type:

```lisp
(sql:select [MyDate] :from [MyTable] :where [= [id] 1])
```

#### 23.6.2.1 Timezone of returned DATEs

Common SQL creates universal time values from DATE fields assuming that the database contains times in Coordinated Universal Time (UTC). That is, as if by passing `time-zone` 0 to `encode-universal-time`. To decode the values consistently with this encoding, pass `time-zone` 0 to `decode-universal-time`.

If the database contains times in a different timezone, then the integer `time-zone` needs to be adjusted by adding an appropriate multiple of 3600 before calling `decode-universal-time`.

### 23.6.3 DATE returned as string

Instead of universal time integers, you can obtain strings formatted by the database by modifying the `MyDate` database identifier, adding `:string` like this:

```lisp
(sql:select [MyDate :string] :from [MyTable] :where [= [id] 1])
```

This avoids the overhead of converting DATEs to universal times and so may improve performance of your application.

See `select`, page 1634 for details.
23.6.4 Using universal time format

If the database is only accessed via Common SQL and you want to use the universal time date format, then you might consider using an INTEGER column containing universal time values instead of a DATE column.

23.7 SQL I/O recording

It is sometimes convenient to simply monitor the flow of commands to, and results from, a database. A number of functions are provided for this purpose.

The functions operate on two stream collections (*broadcast streams*) — one each for commands and results. They allow the recording to be started and stopped, checked, or recorded on further individual streams. By default, both commands and results recording is printed only to *standard-output*.

For details, see the reference pages for `start-sql-recording`, `stop-sql-recording`, `sql-recording-p`, `list-sql-streams`, `sql-stream`, `add-sql-stream` and `delete-sql-stream`.

23.8 Error handling in Common SQL

All errors generated by Common SQL are of type `sql-user-error` or `sql-database-error`. You can test for these conditions and their subtypes in your error handlers.

23.8.1 SQL condition classes

An `sql-user-error` is an error inside Lisp.

An `sql-database-error` is an error inside the database interface that Lisp uses.

The following are subclasses of `sql-database-error`:

- `sql-database-data-error` is an error with the data given. It signifies an error that must be fixed for the code to work.

- `sql-timeout-error`
Signifies an error that is a result of other users using the same database. It means the code can work without change, once the other users stop using the database.

**sql-connection-error**

An error with the connection to the RDBMS.

The following are subclasses of **sql-connection-error**:

- **sql-timeout-error**
  
  A timeout with some operation.

- **sql-fatal-error**
  
  An error which means that the connection is no longer usable.

**Note:** In general, the documentation for the various supported databases make it difficult to decide which error code should be made into which of the above condition class, and we probably get many of these wrong. If you find errors that seem to be signaled with the wrong condition class, please report them to Lisp Support, including the full printout of the condition, and we will fix it.

### 23.8.2 Database error accessors

Three functions are provided which access slots of **sql-database-error**, allowing you to discover more about the actual error that occurred.

**sql-error-error-id** and **sql-error-secondary-error-id** return primary and secondary error identifiers. If you use these, please read the detailed description in **sql-database-error**, page 1643.

**sql-error-database-message** is a string (maybe nil) returned by the foreign code.

### 23.9 Using MySQL

This section describes particular issues in Common SQL with MySQL databases.
23.9.1 Connection specification

See “Connecting to MySQL” on page 339 for information about MySQL specific extensions for the connection-spec passed to connect.

23.9.2 Case of table names and database names

MySQL is case sensitive on table names and database names when the server is on a Unix machine. MySQL does not automatically change raw names to uppercase as specified by the SQL standard. However, Common SQL is geared towards uppercasing all names, so this may cause some mismatches. In general, Common SQL uppercases strings, and uses symbol names, which are normally uppercase, as-is.

One solution, possible only if you control the naming of tables and databases, is to make them all have the same case. If this is uppercase, that suffices. If it is lowercase, you need to set the variable lower_case_table_names in the configuration of the server.

If you cannot make all the names the same case, you have to get the case right. This can be achieved in several ways:

1. Specify tables names using strings, for example:

   (sql:select [*] :from "TableNAMEwithVARIABLEcase")

   Note that this does not work in LispWorks 4.4 and previous versions.

2. Pass the Lisp string directly:

   (sql:select [*] :from "TableNAMEwithVARIABLEcase")

   Note that in this case the table name is passed to the database inside double quotes. That works only when the mode of the Common SQL connection contains ANSI_QUOTES (which is the default, see “SQL mode” on page 375 for details).

3. Specify table names as escaped symbols:

   (sql:select [*] :from [\|TableNAMEwithVARIABLEcase|])

4. Construct the whole query string and pass it to query rather than using select:

   (sql:query "select * from TableNAMEwithVARIABLEcase")
23.9.3 Encoding (character sets in MySQL).
You can specify the encoding to be used by passing the :encoding argument to connect. Common SQL supports various encodings for MySQL as documented in connect, page 1533.
The default is to use the default for the particular MySQL installation.

23.9.4 SQL mode
Because Common SQL is geared towards ANSI SQL, by default it connects in ANSI mode. If another mode is required, it can be set at connection time.
For example, to make MySQL treat quotes as in ANSI without setting other ANSI features, do:

```sql
(sql:connect "me/mypassword/mydb"
 :sql-mode "ANSI_QUOTES")
```
See the description of the :sql-mode argument to connect, page 1533 for details.

23.9.5 Meaning of the :owner argument to select
In the Common SQL MySQL interface, the value of the select keyword argument :owner is interpreted to select a database name.

23.9.6 Special considerations for iteration functions and macros
This section describes particular issues when fetching multiple records using Common SQL with MySQL databases.

23.9.6.1 Fetching multiple records
The function map-query and the macros do-query, simple-do-query and loop with each record use internally mysql-use-query, which means that the underlying MySQL code brings the data from the server one record at a time. With a small number of records, it may be preferable to bring all the data immediately instead. This can be done by passing the argument get-all, as follows:
(sql:map-query nil 'print
   "select forname, surname from people"
   :get-all t)

(sql:do-query
  ((forname surname) "select forname, surname from people"
   :get-all t)
   body)

(sql:simple-do-query
  (list "select forname, surname from people"
       :get-all t)
   body)

(loop for (forname surname) being each record
   "select forname, surname from people"
   get-all t
   body)

23.9.6.2 Aborting queries which fetch many records

In the MySQL interface there is no way to abort a query when part way through it. When any of the iterations above stops before reaching its end, the underlying code retrieves all the records to the end of the query (though without converting them to Lisp objects). If the query found many records, that may be an expensive (that is, time consuming) operation.

It is possible to avoid this inefficiency by passing the argument not-inside-transaction. If not-inside-transaction is true then when a query is aborted, then LispWorks closes the database connection and reopens it, rather than retrieving all the remaining records.

(sql:map-query nil 'print
   "select forname, surname from people"
   :get-all t
   :not-inside-transaction t)

Note that this will lose any state associated with the connection, and so not-inside-transaction should only be used with care.

23.9.7 Table types

By default, create-table creates tables of the default type. This behavior can be overridden by the connect keyword arguments :default-table-type

If type is passed to create-table or default-table-type was passed to connect, it is used as the argument to the "keyword" TYPE in the SQL statement:

```
create table MyTable (column-specs) TYPE = type-value
```

If extra-options is passed to create-table or default-table-extra-options was passed to connect, it is appended in the end of the SQL statement above.

connect with default-table-type and create-table with type also accept the keyword argument :support-transactions. When support-transactions is true, these functions will attempt to make tables that support transactions. It does this by using the type innodb.

### 23.9.8 Rollback errors

The default value of the connect keyword argument :signal-rollback-errors is determined by the value of the :default-table-type argument. If default-table-type is :support-transactions or "innodb" or "bdb", then the default value for :signal-rollback-errors is t, otherwise the default value is nil.

### 23.9.9 Types of values returned from queries

Common SQL uses the MySQL mechanism that returns values as strings.

By default, Common SQL converts these strings to the appropriate Lisp type corresponding to the column type (or more accurately, the type of the field in the query) according to Table 23.3

<table>
<thead>
<tr>
<th>MySQL column type</th>
<th>Lisp Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>All integer types</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>double-float</td>
<td></td>
</tr>
<tr>
<td>Single</td>
<td>single-float</td>
<td></td>
</tr>
</tbody>
</table>
However, if you specify the result type as :string, this eliminates the conversion and the return value is simply the string retrieved by MySQL. For information about specifying the result type for a column (or multiple columns) in a query, see “Querying” on page 346.

Each of the five date-like types (that is, Date, Datetime, Timestamp, Time and Year) can have result type :date, :date-string or :datetime-string with the following effects:

- **:date**
  This result type means a Universal time. This is the default except for Year.

- **:date-string**
  A string with the format that MySQL uses for Date columns.

- **:datetime-string**
  A string with the format that MySQL uses for Datetime columns.
All the numeric types can have result type `:int`, `:single-float` or `:double-float`, causing the appropriate conversion. No check is made on whether the result is actually useful.

String types can have result type `:binary`, which returns an array.

### 23.10 Using Oracle

This section describes particular issues in Common SQL with Oracle databases, apart from the LOB interface, which is described in “Oracle LOB interface” on page 380.

#### 23.10.1 Connection specification

See “Connecting to Oracle” on page 338 for information about Oracle-specific interpretation of the `connection-spec` passed to `connect`.

#### 23.10.2 Setting connection parameters

Oracle database connections have prefetch values which you can control via Common SQL. Alternatively you can allow the database default prefetch values to take effect.

You can set the default prefetch values for a connection by passing `:prefetch-rows-number` and `:prefetch-memory` keyword arguments to `connect`. The default value of `prefetch-rows-number` is 100 and the default value of `prefetch-memory` is `#x100000` (meaning 1 MB of data).

You can also pass the value `:default` for either of these arguments. This means that Common SQL does not set the default. This is useful if Oracle itself provides a suitable default.
23.11 Oracle LOB interface

23.11.1 Introduction

The Common SQL Oracle LOB interface allows you to retrieve LOB locators and then perform operations on them. It is also possible to insert new empty LOBs.

23.11.1.1 Retrieving LOB locators

This is done by normal select or query calls where the selections list names one or more columns that are of a LOB type. The LOB types are BLOB, CLOB, NCLOB, BFILE and CFILE.

The returned value is a LOB locator: an opaque Lisp object on which the ora-lob-* APIs (that is, those functions with names beginning with "ora-lob-") can be used. This LOB locator contains a pointer to an Oracle descriptor of type OCILobLocator*. Note that there can be multiple LOB locator objects associated with the same LOB in the server, but a LOB locator uniquely identifies a LOB object.

It is possible to specify that the result object should be a stream either for input or output. Then the resulting stream (which will be of type lob-stream) can be used as a normal Lisp stream.

23.11.1.2 Operating on LOB locators

This is done using the ora-lob-* functions. Most of these functions map directly to the underlying OCILob* functions.

Note that when modifying a LOB locator, the corresponding record must be locked. See “Retrieving Lob Locators” on page 381 for details.

23.11.1.3 Inserting empty LOBs

To add a new LOB object to the database, you must insert an empty LOB. The preferred way of doing this is to use the Oracle SQL functions EMPTY_BLOB and EMPTY_CLOB, which can called by using the pseudo operator sql-function, like this:
23.11 Oracle LOB interface

```
(sql:insert-records :into [mytable]
  :values
  (list "name" [sql-function 'empty_blob]))
```

This code inserts a record with "name" and an empty BLOB. It is also possible to make an empty LOB by calling `ora-lob-create-empty`, and passing the empty LOB as a value to `insert-records` or `update-records`.

23.11.2 Retrieving Lob Locators

When the selections list of a query that is used in `select`, `query`, `do-query`, `map-query`, `simple-do-query` or `loop .... for x being each record` contains a column of a LOB type, the results are LOB locator objects. For example, if the table definition is:

```
create table mytable {
  name varchar(200),
  image  blob
}
```

Then doing

```
(sql:select [image] :from [mytable] :flatp t)
```

returns a list of LOB locators.

This example lists the size of the images in the table mytable:

```
(dolist (pair (sql:select [name][image] :from [mytable]))
 (format t "-a has an image of size ~a~"
        (first pair) (sql:ora-lob-get-length (second pair))))
(sql:ora-lob-free (second pair))
```

or more efficiently

```
(sql:do-query ((name lob-locator)
  [sql:select [name][image] :from [mytable]])
  (format t "-a has an image of size ~a~"
        name (sql:ora-lob-get-length lob-locator)))
```

**Note:** The lifetime of the LOB locator objects differs between the functions that return a list of objects (`select` and `query`) and the iterative functions and macros (`do-query`, `simple-do-query`, `loop` and `map-query`). The iteration functions and macros free the LOB locators that they retrieve before proceeding to the next iteration. `select` and `query` do not free the LOB locators. Each
LOB locator stays alive until the application makes an explicit call to `ora-lob-free`, or until the database is closed by a call to `disconnect`.

### 23.11.3 Locking

When the LOB or its contents need to be modified, the corresponding record must be locked (Oracle enforces this). The best way to lock a record is to pass `:for-update` when calling `select`. See `select`, page 1634 for details. For example, writing a line in the end of the log file of station number 573:

```sql
create table logfiles (stationid integer, logfiles clob)
.. insert records ..
(sql:do-query ((log-stream)
    [select [log :output-stream] :from [logfiles]
(file-position log-stream :end)
(write-line "Add this line to the log" log-stream)
(close log-stream) ; forces the output
)
(sql:commit)
```

Note that any call to `commit` or `rollback` on the same connection removes the lock. If you want to modify the LOB later, you must lock it again. An efficient way to achieve this is to use the special token `ROWID`, which returns the `ROWID` in the database, because this does not involve searching on the server side. For example:
23.11 Oracle LOB interface

(let ((lobs-list
  (sql:select [lob-field] [rowid] ; get pairs of LOB
            :from [mytable] ; locators and ROWIDs
            :where [some-condition]))))

... do something ...

... reach a point when we want to modify one
... of the LOBS above and have bound one of the
... pairs in the variable pair.

(sql:select [1]
  :from [mytable] ; retrieve a constant
  :where
  [= [rowid] (second pair)] ; get the right record
  :for-update t) ; lock it

(sql:ora-lob-write-buffer (car pair) ; modify the lob
  offset
  amount
  buffer)

(sql:commit) ; also unlock everything
)

23.11.4 Retrieving LOB Locators as streams

To retrieve LOB locators as streams, specify the type of retrieved object as :input-stream or :output-stream in the query. For example:


returns a list of streams.

For example, to print the name of all images that start with some "magic number", that is a sequence of 4 specific bytes (#xf5 #x12 #x4e #x23):

(let ((array (make-array 4 :element-type '(unsigned-byte 8))))
  (sql:do-query ((name lob-stream)
    [sql:select [name][image :input-stream]
      :from [mytable]])
    (when (and (eq (read-sequence array lob-stream) 4)
      (eq (aref array 0) #xf5)
      (eq (aref array 0) #x12)
      (eq (aref array 0) #x4e)
      (eq (aref array 0) #x23))
      (print name)))))

Closing the stream also frees the LOB object.

When using :output-stream, it is important to call force-output before trying to commit the changes, because the stream is buffered.
23.11.5 Attaching a stream to a LOB locator

It is possible to attach a stream to a LOB locator, passing the LOB locator as a :lob-locator argument to (make-instance 'lob-stream ...). The value of the :direction argument must be :input or :output. By default, if the stream is closed the LOB locator is freed, unless the value of the initarg :free-lob-locator-on-close is passed as nil.

Operations via the stream can be mixed with direct operations on the LOB. However, because of the buffering, accessing the LOB contents will give non-obvious results, as other operations may not see something that was written to the stream because it is still in the stream buffer, or the stream may have already read some contents before they were overwritten. Use force-output or clear-input before accessing the LOB in other ways to avoid these problems.

It is possible to attach more than one stream to the same LOB locator, in both directions. Apart from the issue of the buffering described above, the streams can be used independently of each other. Note that if you want to close one of the streams and to continue to use the others or the LOB locator itself, you must pass :free-lob-locator-on-close nil when you make the stream.

The LOB locator to which a stream is attached can be found by using the reader lob-stream-lob-locator.

23.11.6 Interactions with foreign calls

You can define your own foreign calls and use them on the underlying OCI descriptors. For this, you need to access the OCI handles using ora-lob-lob-locator, and maybe ora-lob-env-handle and ora-lob-svc-ctx-handle. These accessors return foreign pointers that can be passed to foreign functions in the usual way.

When the foreign functions deal only with the data, rather than with LOB objects, use the functions ora-lob-read-foreign-buffer, ora-lob-write-foreign-buffer and ora-lob-get-buffer.

For example:
You have a C function `my_lob_processor`

```lisp
(fli:define-foreign-function my-lob-processor
 (lob (sql:p-oci-lob-locator)
        (env (sql:p-oci-svc-ctx))
        (other-arg :int))
 :result-type :int)
```

Assuming you have the LOB locator in the variable `lob`, call the foreign function on it:

```lisp
(my-lob-processor (sql:ora-lob-lob lob)
                   (sql:ora-lob-svc-ctx-handle lob)
                   36)
```

There are three handles in the LOB: the LOB descriptor itself, the environment and the context. The pointer types, the reader and the corresponding C type for each handle are shown in Table 23.4 below.

<table>
<thead>
<tr>
<th>OCI handle</th>
<th>Reader</th>
<th>Pointer type</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOB descriptor</td>
<td>ora-lob-lob-locator</td>
<td>p-oci-lob-locator</td>
<td>OCILobLocator*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or p-oci-file</td>
<td></td>
</tr>
<tr>
<td>context</td>
<td>ora-lob-svc-ctx-handle</td>
<td>p-oci-svc-ctx</td>
<td>OCISvcCtx*</td>
</tr>
<tr>
<td>environment</td>
<td>ora-lob-env-handle</td>
<td>p-oci-env</td>
<td>OCIEnv*</td>
</tr>
</tbody>
</table>

The pointer type `p-oci-lob-locator` is used for internal LOBs (that is, BLOB, CLOB and NCLOB). The pointer type `p-oci-file` is used for file LOBs (CFILE and BFILE). For functions that take both, the type `p-oci-lob-or-file` is defined as the union of these two types.
23.11.7 Determining the type of a LOB

The function `ora-lob-internal-lob-p` returns whether it is internal (that is BLOB, CLOB or NCLOB) or not (that is BFILE or CFILE). The function `ora-lob-element-type` returns the LISP element type that best corresponds to the LOB locator. This will be one of `(unsigned-byte 8)` for BLOB and BFILE, or `base-char` or `bmp-char` for CLOB, NCLOB and CFILE, depending on the charset of the LOB object.

It is possible to distinguish between CLOB and NCLOB by looking at the result of `ora-lob-char-set-form`. It returns 2 for NCLOB and 1 for CLOB.

23.11.8 Reading and writing from and to LOBs

One way of reading and writing is to use streams as described in the section “Retrieving LOB Locators as streams” on page 383. When large amounts of data are written (read) to (from) the LOB the direct interface may be useful. The direct interface is implemented by `ora-lob-read-foreign-buffer`, `ora-lob-read-buffer`, `ora-lob-write-foreign-buffer`, and `ora-lob-write-buffer`.

All the direct interfaces are more efficient if the buffer that is passed is static. That is always true for the `*-foreign-buffer` functions, but normally not true for Lisp objects. See the documentation for `make-array`, page 680. See also `ora-lob-get-buffer`.

The direct reading and writing methods can be used for “random” access, but they can also be used conveniently for efficient linear access, simply by passing `nil` as the offset parameter.

23.11.9 The LOB functions

Most of the LOB functions take an `errorp` argument, which is a boolean controlling what happens if an error occurs inside an OCI function. If `errorp` is true, an error is signaled. If `errorp` is false, the function returns an error object (of type `sql-database-error`).

All the LOB functions signal an error if the `lob-locator` argument given is not a LOB locator object as returned by `select` or `query`. 
Many of the functions basically perform a call to the underlying OCI function. When the match is direct, this is mentioned in the function’s manual page.

### 23.11.9.1 Querying functions

You can test whether a LOB locator is initialized, open or temporary with `ora-lob-locator-is-init`, `ora-lob-is-open` or `ora-lob-is-temporary`.

The predicate for internal LOBs is `ora-lob-internal-lob-p`.

`ora-lob-element-type` returns a Lisp element type corresponding to the LOB locator as described “Determining the type of a LOB” on page 386.

`ora-lob-lob-locator`, `ora-lob-env-handle` and `ora-lob-svc-ctx-handle` return foreign pointers to the various handles in the LOB mentioned in “Interactions with foreign calls” on page 384. To determine the best value for the size of a buffer use `ora-lob-get-chunk-size`.

`ora-lob-char-set-form` and `ora-lob-char-set-id` query the charset of a lob-locator.

The querying functions specifically for file LOBs are `ora-lob-file-exists`, `ora-lob-file-is-open` and `ora-lob-file-get-name`.

You can obtain the current length of the LOB with `ora-lob-get-length`.

You can test two LOB locators for whether they point to the same LOB object with `ora-lob-is-equal`.

### 23.11.9.2 LOB management functions

You can create a LOB object with `ora-lob-create-empty`.

You can assign a LOB to another LOB locator with `ora-lob-assign`.

You can free a LOB locator with `ora-lob-free`.

### 23.11.9.3 Modifying LOBs

All the functions mentioned in this section are applicable to internal LOBs only, except `ora-lob-load-from-file`. 
Before modifying a LOB, the corresponding record must be locked. See the discussion in “Locking” on page 382.

If you make several modifications to a LOB which has functional or domain indexes, it is useful to wrap several calls of modifying functions in a pair of ora-lob-open and ora-lob-close. That means that the indexes will be updated once (when ora-lob-close is called), which saves work. Note that after a call to ora-lob-open, ora-lob-close must be called before any call to commit.

To append the contents of one LOB to another, use ora-lob-append.

You can copy all or part of a LOB into another LOB using ora-lob-copy.

ora-lob-load-from-file loads the data from a file LOB into an (internal) LOB.

You can erase (that is, fill with the 0 byte or with Space character) all or part of a LOB using ora-lob-erase.

You can reduce the size of a LOB using ora-lob-trim.

If you need to make multiple updates to a LOB you can optionally create a transaction using ora-lob-open and ora-lob-close call. This may save work on the server side.

### 23.11.9.4 File operations

These functions are used to modify the properties of file LOBs.

Open and close the file associated with a file LOB using ora-lob-file-open and ora-lob-file-close.

You can close all the files associated with a file LOB locator that have been opened through the database connection with ora-lob-file-close-all.

You can alter the directory and/or the file name for a file LOB locator by calling ora-lob-file-set-name.

### 23.11.9.5 Direct I/O

The direct I/O functions perform input or output directly on the OCI handle, without the intervening layer of a stream. If you move large amounts of data
to or from the LOB, and in particular if you pass the data to or from foreign functions, the direct calls can be more efficient, and in some cases also more convenient to use. Note, however, that if you make many small modifications to the data, the \texttt{lob-stream} interface may be more efficient.

Note also that the difference in efficiency between the direct calls and the \texttt{lob-stream} interface is likely to be quite small compared to the time spent on network traffic.

If you make many modifications to a LOB, you should also consider wrapping the operations in a transaction created by a pair of calls to \texttt{ora-lob-open} and \texttt{ora-lob-close}.

You can read data from the LOB locator into a Lisp buffer or foreign buffer using \texttt{ora-lob-read-buffer} and \texttt{ora-lob-read-foreign-buffer} respectively.

Similarly \texttt{ora-lob-write-buffer} and \texttt{ora-lob-write-foreign-buffer} can be used to write buffer to a LOB.

You can obtain a buffer suitable for efficient I/O with foreign functions via \texttt{ora-lob-get-buffer}.

\texttt{ora-lob-read-into-plain-file} writes the contents of a LOB into a file.

\texttt{ora-lob-write-from-plain-file} writes the contents of a file into a LOB.

### 23.11.9.6 Temporary LOBs

You can create a temporary LOB with \texttt{ora-lob-create-temporary}.

You can test whether a LOB is temporary with \texttt{ora-lob-is-temporary}.

You can free a temporary LOB locator if necessary with \texttt{ora-lob-free-temporary}, though temporary LOB locators are freed automatically when the database connection is closed by \texttt{disconnect}.

### 23.11.9.7 Control of buffering

These functions control the internal buffering by the Oracle client: \texttt{ora-lob-enable-buffering}, \texttt{ora-lob-disable-buffering}, and \texttt{ora-lob-flush-buffer}. They have no interaction with any of the other functions above.
23.11.10 Fetching the contents of the LOBs directly

Sometimes it useful to fetch the contents of a LOB directly. You can do that by specifying the type of the requested value as \texttt{:binary} for binary LOBs (BLOB and BFILE) or \texttt{:string} for character LOBs (CLOB, NCLOB, and CFILE). When you specify the type in this way, the fetched values are arrays of type \texttt{(unsigned-byte 8)} for \texttt{:binary} and strings for \texttt{:string}. For example:

\begin{verbatim}
(sql:select [blob_column] :from [a_table]) =>
a list of LOB locators

(sql:select [blob_column :binary] :from [a_table]) =>
a list of arrays
\end{verbatim}

23.12 Using ODBC

23.12.1 Configuring unixODBC

On Unix, configure unixODBC in these files.

For the driver:

/etc/odbcinst.ini

For the datasource:

~/.odbc.ini

/etc/odbc.ini

23.12.2 Loading unixODBC

At load time do:

(require "odbc")

At run time on Unix, Common SQL automatically loads the unixODBC module from the location in the variable \texttt{sql::*odbc-foreign-modules*}. In LispWorks for Linux this variable initially has the value \texttt{"/usr/lib/libodbc.so"}. Therefore if, for example, the run time machine unixODBC
23.12 Using ODBC

installed in
/usr/local/, at run time do:

(setq sql::*odbc-foreign-modules* '(*user/local/lib/libodbc.so*))
(sql:connect "mydatabase" :database-type :odbc)

23.12.3 External format for ODBC strings

On Unix, the default external format for ODBC strings is :ascii. On Microsoft Windows it is win32:*multibyte-code-page-ef*.

23.12.4 Using non-ASCII strings on Microsoft SQL Server

When passing a SQL expression containing string literals to Microsoft SQL Server (which you can do via ODBC), if a string literal contains characters that the server’s code page cannot represent, then the string literal needs to be marked as "Native" by prefixing it with the character 'N' before the opening quote. For example:

N'Greek'

Code pages always can always represent ASCII characters, but differ in what other characters can represent. The functions string-needs-n-prefix and string-prefix-with-n-if-needed are provided to check if a string needs prefixing.

Other SQL backends work with all strings regardless of the N syntax, but the syntax is allowed by most of them as well (and is standard SQL). However, SQLite and Microsoft Access (via ODBC) do not recognize the N syntax, and give an error. This means that static SQL expressions, which are generated before knowing which SQL backend is going to be used, cannot reliably use the N syntax. In addition, knowing exactly which strings need the N syntax requires knowledge of the code page in the server, and hence requires the database to be opened already when string-needs-n-prefix or string-prefix-with-n-if-needed are called.

The syntax described in “Symbolic SQL syntax” on page 358 generates static expressions when possible, and Lisp string values within them are processed independently of any database to produce string literals without the N syntax. This can be overridden by using the string pseudo-operator, which is described in “SQL string literals” on page 366, and can decide dynamically
whether to use the N syntax or not. Thus you should use the string pseudo-operator in any symbolic SQL syntax that may be used with Microsoft SQL Server and contains SQL string literals (including Lisp expressions that evaluate to strings) to ensure that it works on Microsoft SQL Server for all strings and but is also portable.

If you want to work with Microsoft SQL Server and do not require portability to SQLite or Microsoft Access, then you can set *use-n-syntax-for-non-ascii-strings* to t to always use the N syntax. However, the N prefix changes the type of the string inside Microsoft SQL Server to "Unicode", which has a different collation to non-Unicode strings, so if you need the non-Unicode collation for strings that have codes in the server’s code page then this may not be the right approach.

Another approach is to use prepare-statement with a bind-variable for the string, which works on all SQL backends without any additional code (because the string is not used as a literal in the SQL expression):

```lisp
(setq *a-prepared-statement*
  (sql:prepare-statement [sql:select [name]
    :from [sometable]
    :where [= [nchar_column] [1]]]))

...(sql:set-prepared-statement-variables *a-prepared-statement*
  (list a-non-ascii-string))
(sql:query *a-prepared-statement*)
```

The functions update-records and insert-records also do not use the values that they get as literals in SQL expressions when modifying a Microsoft SQL Server database, and therefore do not require additional code for the values. However, the where expression in update-records and the query expression in insert-records are used directly, so if they contain non-ASCII strings as literals then they will need to be modified for Microsoft SQL Server.

### 23.13 Using SQLite

This section describes particular issues in Common SQL with SQLite databases.
23.13 Using SQLite

23.13.1 Connecting to SQLite

See “Connecting to SQLite.” on page 343 for information about SQLite-specific connection-spec and sqlite-keywords arguments to connect.

23.13.2 Types of retrieved fields in queries

By default, when doing queries (select, query, map-query, do-query, simple-do-query, loop with each record and print-query) the LispWorks checks the data type of each field it reads in each row, and fetches the data accordingly (using the C functions sqlite3_column_* like sqlite3_column_int in the SQLite3 library). Values of SQLite data types NULL, INTEGER, REAL and TEXT are mapped to Lisp objects of type null, integer, double-float and string respectively (“mapped” means returned from select or query, printed in print-query, or passed to your code in the other APIs). A value of data type BLOB is mapped to an array with element type (unsigned-byte 8) containing all of the bytes of the BLOB.

You can force the value to a specific type by specifying the type explicitly. This is done by specifying the type with the identifier, either using the symbolic SQL syntax (see “Enclosing database identifiers” on page 359) or using sql-expression. For select and query you can also use the keyword argument :result-types.

The types that LispWorks recognizes for SQLite are the common types: :integer (alias :int), :string, :double-float and :binary. These match the SQLite data types INTEGER, TEXT, REAL and BLOB respectively. When these keywords are used, LispWorks asks SQLite for a value of the corresponding data type, and converts it to the matching Lisp object type as above. Note that the value can also be nil, if the the value is null or cannot be converted to the requested Lisp object type.

Other possible values for the type are:

: single-float

LispWorks asks SQLite for a REAL, and coerces it to a single-float.
23 Common SQL

nil Use the default behavior. Useful if you use :result-types and want to force the type of some of the fields but not all of them.

:blob Returns a handle to the raw data of a BLOB, from which you can read the data using the APIs described in “Reading from blobs using a handle (sqlite-raw-blob) and modifying blobs (sqlite-blob)” on page 394. This allows more flexible access to BLOB values.

:blob cannot be used with select or query.

Note that SQLite does not support any kind of date data type.

When the value that is stored in the database does not match the value that it is asked for, the SQLite library converts the value to the required type, so you always get a value of the correct type, but not necessarily a useful value. See the documentation for SQLite for details: https://www.sqlite.org/c3ref/column_blob.html "Result Values From a Query".

23.13.3 Tables containing a uniform type per column

SQLite allows the fields in each row to contain any supported type, rather than being constrained to the type specified for the column in the table definition.

When you connect to a database, you can use the SQLite-specific keyword :uniform-type-per-column in sqlite-keywords with value t to tell LispWorks that all of the values of a column returned by a query have the same data type.

When you do that, for fields where you do not specify the type explicitly, the LispWorks checks the type of the field in the first result row, and then uses it for the rest of the rows.

23.13.4 Reading from blobs using a handle (sqlite-raw-blob) and modifying blobs (sqlite-blob)

When the type of a field in a query is specified as :blob, the SQLite BLOB is mapped to an object of type sqlite-raw-blob. You can then read data from the SQLite BLOB using any the functions copy-from-sqlite-raw-blob,
23.13 Using SQLite

replace-from-sqlite-raw-blob or sqlite-raw-blob-ref. The function sqlite-raw-blob-length can be used to find the size of the BLOB (in bytes).

The sqlite-raw-blob is valid only within the dynamic extent of the function that is called from the Common SQL interface. If you try to read from a sqlite-raw-blob outside this dynamic context, an error of type sql-user-error will be signaled. You can use sqlite-raw-blob-valid-p to check if a blob is valid.

Using sqlite-raw-blob makes it more convenient to read the data when a BLOB contains elements larger than bytes, and makes it more efficient when you retrieve large BLOBs (a few kilobytes or more) but need only a small part of the data.

SQLite allows reading and writing of BLOBs (fields with type BLOB or TEXT) directly, which you can do using the sqlite-blob interface. The functions sqlite-open-blob and sqlite-close-blob are used to open and close a BLOB field, or the macro with-sqlite-blob can be used to do both. Once you have opened a BLOB, you call replace-from-sqlite-blob or replace-into-sqlite-blob to copy data to or from it. Note that the sqlite-blob is not thread-safe, so you must do all of the operations in a “single thread” context (either all in one thread, or serialized by a lock).

sqlite-raw-blob corresponds to the result of the C function sqlite3_column_blob (and sqlite3_column_bytes to obtain the size). sqlite-blob corresponds to the C structure sqlite3_blob.

23.13.5 Values in Insert and Update.

When modifying a table in SQLite, either directly by using insert-records or update-records, or by executing a prepared-statement statement with bind-variables, the values that are passed are treated as follows:

In a prepared-statement, if the variable-type is :string, then the value is converted to a string.

The value is passed to the SQLite library as a SQLite data type based on the type of the value as follows:

(signed-byte 64)

INTEGER
A binary array is an array with an integer or float element type. `bmp-string` and `base-string` are also binary arrays in some contexts, but they are treated as strings in this case (`text-string` is not a binary array).

In addition, the value can be a list, which in treated as follows:

- If the first element of the list is a binary array, including `bmp-string` or `base-string`, then the list must be of form `(array)`, `(array start)` or `(array start end)` and the bytes between `start` and `end` in `array` are inserted as a BLOB. If `start` is omitted, it defaults to 0. If `end` is omitted, it defaults to the length of `array`.

  Note that `start` and `end` are denoted in elements rather than bytes, so the number of the bytes in the BLOB is `(* (- end start) bytes-per-element)`.

  Note also that, for arrays of more than one byte per element, the contents of the BLOB will depend on the byte order of the host machine.

  Apart from allowing insertion of parts of arrays, this syntax also allows you to insert the character codes in a `bmp-string` and `base-string` as a BLOB, by passing the string as a list of one element.

- If the first element of the list is the keyword `:zeroblob`, then the second element is treated as a size, which must be a positive integer smaller than $2^{31}$, that is of type `(integer 0 #x7fffffff)`. LispWorks inserts a zero blob of this size (using the C function `sqlite3_bind_zeroblob`).

Any value that does not match the description above, including integers out of range and lists that do not match the patterns described, cause an error (of type `sql-user-error`) to be signaled.

### 23.13.6 Accessing ATTACHed databases

ATTACHed databases in SQLite, that is databases that were attached using the SQLite ATTACH statement, are identified by their schema names. You can
specify the schema name in the "[...]" syntax, for example, if you attach a file called "another-database" as follows:

```
(execute-command "ATTACH another-database as AttachedDB")
```

then you can read the contents of a table SomeTable inside "another-database" using AttachedDB as the "schema" name:

```
(select [*] :from [AttachedDB SomeTable])
```

The keyword `:owner` in Common SQL function specifies the schema to which the table(s) belong, for example, after the ATTACH above, you can obtain the list of tables inside another-database by using:

```
(sql:list-tables :owner "AttachedDB")
```

and use AttachedDB as the "owner" in sql-expression:

```
(select [*] :from (sql-expression :owner "AttachedDB" :table "SomeTable"))
```

See https://www.sqlite.org/lang_attach.html "ATTACH DATABASE" for details about attaching in SQLite.
23 Common SQL
24

User Defined Streams

24.1 Introduction

A number of classes and functions are provided in the stream package that allow you to define your own input and output streams. You can use the standard Common Lisp I/O functions on these streams, and you can add methods specialized on your stream classes to provide specific implementations of other I/O functions. Note that some changes have been made to the standard I/O functions to allow for this. For example, stream-element-type is now a generic function. See Chapter 33, “The COMMON-LISP Package” for alterations to Common Lisp functions, and Chapter 48, “The STREAM Package” for more details on the API for user defined streams.

24.2 An illustrative example of user defined streams

In this chapter an example is provided to illustrate the main features of the stream package. In this example a stream class is defined to provide a wrapper for file-stream which uses the Unicode Line Separator instead of the usual ASCII CR/LF combination to mark the end of lines in the file. Methods are then defined, specializing on the user defined stream class to ensure that it handles reading from and writing to a file correctly.
24.2.1 Defining a new stream class

Streams can be capable of input or output (or both), and may deal with characters or with binary elements. The stream package provides a number of stream classes with different capabilities from which user defined streams can inherit. In our example the stream must be capable of input and output, and must read characters. The following code defines our stream class appropriately:

```lisp
(defclass unicode-ls-stream
  (stream:fundamental-character-input-stream
   stream:fundamental-character-output-stream)
  ((file-stream :initform nil
    :initarg :file-stream
    :accessor ls-stream-file-stream)))
```

The new class, `unicode-ls-stream`, has `fundamental-character-input-stream` and `fundamental-character-output-stream` as its superclasses, which means it inherits the relevant default character I/O methods. We shall be overriding some of these with more relevant and efficient implementations later.

Note that we have also provided a `file-stream` slot. When making an instance of `unicode-ls-stream` we can create an instance of a Common Lisp file stream in this slot. This allows us to use the Common Lisp file stream functionality for reading from and writing to a file.

24.2.2 Recognizing the stream element type

We know that the stream will read from a file using `file-stream` functionality and that the stream element type will be `character`. The following defines a method on `stream-element-type` to return the correct element type.

```lisp
(defmethod stream-element-type ((stream unicode-ls-stream))
  'character)
```

24.2.3 Stream directionality

Streams can be defined for input only, output only, or both. In our example, the `unicode-ls-stream` class needs to be able to read from a file and write to a file, and we therefore defined it to inherit from an input and an output stream class. We could have defined disjoint classes instead, one inheriting
from *fundamental-character-input-stream* and the other from *fundamental-character-output-stream*. This would have allowed us to rely on the default methods for the direction predicates.

However, given that we have defined one bi-directional stream class, we must define our own methods for the direction predicates. To allow this, the Common Lisp predicates *input-stream-p* and *output-stream-p* are implemented as generic functions.

```lisp
(defun input-stream-p (stream)
  (input-stream-p (ls-stream-file-stream stream)))

(defun output-stream-p (stream)
  (output-stream-p (ls-stream-file-stream stream)))
```

The above code allows us to “trampoline” the correct direction predicate functionality from *file-stream*, using the *ls-stream-file-stream* accessor we defined previously.

### 24.2.4 Stream input

The following method for *stream-read-char* reads a character from the stream. If the character read is a `#
Line-Separator`, then the method returns `#
Newline`, otherwise the character read is returned. *stream-read-char* returns :eof at the end of the file.

```lisp
(defun stream:stream-read-char (stream)
  (let ((char (read-char (ls-stream-file-stream stream)
                         nil :eof)))
    (if (eql char #
Line-Separator)
        #
Newline
       char)))
```

There is no need to define a new method for *stream-read-line* as the default method uses *stream-read-char* repeatedly to read a line, and our implementation of *stream-read-char* ensures that this will work.

We also need to make sure that if a `#
Newline` is unread, it is unread as a `#
Line-Separator`. The following method for *stream-unread-char* uses the Common Lisp file stream function *unread-char* to achieve this.
\begin{verbatim}
(defmethod stream:stream-unread-char ((stream unicode-ls-stream) char)
   (unread-char (if (eql char #\Newline) #\Line-Separator char)
                   (ls-stream-file-stream stream)))
\end{verbatim}

Finally, although the default methods for \texttt{stream-listen} and \texttt{stream-clear-input} would work for our stream, it is faster to use the functions provided by \texttt{file-stream}, again using our accessor \texttt{ls-stream-file-stream}.

\begin{verbatim}
(defmethod stream:stream-listen ((stream unicode-ls-stream))
   (listen (ls-stream-file-stream stream)))
\end{verbatim}
\begin{verbatim}
(defmethod stream:stream-clear-input ((stream unicode-ls-stream))
   (clear-input (ls-stream-file-stream stream)))
\end{verbatim}

\subsection*{24.2.5 Stream output}

The following method for \texttt{stream-write-char} uses \texttt{write-char} to write a character to the stream. If the character written to \texttt{unicode-ls-stream} is a #\Newline, then the method writes a #\Line-Separator to the file stream.

\begin{verbatim}
(defmethod stream:stream-write-char ((stream unicode-ls-stream) char)
   (write-char (if (eql char #\Newline) #\Line-Separator char)
                (ls-stream-file-stream stream)))
\end{verbatim}

The default method for \texttt{stream-write-string} calls \texttt{stream-write-char} repeatedly to write a string to the stream. However, the following is a more efficient implementation for our stream.
(defmethod stream:stream-write-string ((stream unicode-ls-stream) string &optional (start 0) (end (length string)))
  (loop with i = start
        until (>= i end)
        do (let* ((newline (position #\Newline string :start i :end end))
                   (this-end (or newline end)))
            (write-string string (ls-stream-file-stream stream) :start i :end this-end)
            (incf i this-end)
            (when newline
              (stream:stream-terpri stream)
              (incf i)))
  finally (return string)))

We do not need to define our own method for stream-terpri, as the default uses stream-write-char, and therefore works appropriately.

To be useful, the stream-line-column and stream-start-line-p generic functions need to know the number of characters preceding a #\Line-Separator. However, since the LispWorks file stream records line position only by #\Newline characters, this information is not available. Hence we define the two generic functions to return nil:

   (defmethod stream:stream-line-column ((stream unicode-ls-stream)) nil)

   (defmethod stream:stream-start-line-p ((stream unicode-ls-stream)) nil)

Finally, the methods for stream-force-output, stream-finish-output and stream-clear-output are “trampolined” from the standard force-output, finish-output and clear-output functions.

   (defmethod stream:stream-force-output ((stream unicode-ls-stream))
    (force-output (ls-stream-file-stream stream)))

   (defmethod stream:stream-finish-output ((stream unicode-ls-stream))
    (finish-output (ls-stream-file-stream stream)))
(defmethod stream::stream-clear-output ((stream
  unicode-ls-stream))
  (clear-output (ls-stream-file-stream stream)))

24.2.6 Instantiating the stream

Now that the stream class has been defined, and all the methods relevant to it
have been set up, we can create an instance of our user defined stream to test
it. The following function takes a filename and optionally a stream direction
as its arguments and makes an instance of unicode-ls-stream. It ensures
that the file-stream slot of the stream contains a Common Lisp file-
stream capable of reading from or writing to a file given by the filename argu-
ment.

(defun open-unicode-ls-file (filename &key (direction :input))
  (make-instance 'unicode-ls-stream :file-stream
    (open filename
      :direction direction
      :external-format :unicode
      :element-type 'character)))

The following macro uses open-unicode-ls-stream in a similar manner to
the Common Lisp macro with-open-file:

(defmacro with-open-unicode-ls-file ((var filename
  &key (direction :input))
  &body body)
  `(let ((,var (open-unicode-ls-file ,filename
    :direction ,direction)))
    (unwind-protect
      (progn ,@body)
      (close ,var))))

We now have the required functions and macros to test our user defined
stream. The following code uses config.sys as a source of input to an
instance of our stream, and outputs it to the file unicode-ls.out, changing
all occurrences of #\Newline to #\Line-Separator in the process.
(with-open-unicode-ls-file (ss "C:\unicode-ls.out"
   :direction :output)
  (write-line "*- Encoding: Unicode; -*-" ss)
  (with-open-file (ii "C:\config.sys") ; Don't edit this file!
    (loop with line = nil
      while (setf line (read-line ii nil nil))
      do (write-line line ss)))))

After running the above code, if you load the file C:\unicode-ls.out into
an editor (for example, a LispWorks editor), you can see the line separator
used instead of CR/LF. Most editors do not yet recognize the Unicode Line
Separator character yet. In some editors it appears as a blank glyph, whereas
in the LispWorks editor it appears as "<2028". In LispWorks you can use Alt+X
What Cursor Position or Ctrl+X = to identify the unprintable characters.

You can also use the follow code to print out the contents of the new file line
by line.

(with-open-unicode-ls-file (ss "C:\unicode-ls.out")
  (loop while (when-let (line (read-line ss nil nil))
    (write-line line))))
24 User Defined Streams
The interface for using sockets in LispWorks is in the "comm" module, and all the symbols are in the comm package, and documented in the “The COMM Package” on page 513.

To use it you need to require the module by

```
(require "comm")
```

### 25.1 Running a server that accepts connections

The function **start-up-server** starts a new thread which:

1. Creates a socket, then
2. Prepares it (that is, binds it to the address and port and does various other settings) and then
3. Waits for connections to it ("accepting connections")

When a connection is made, a programmer-supplied function is called with the new socket. Typically this function create a stream of type **socket-stream** with this socket, and then uses the stream for communication through the socket using standard Common Lisp I/O functions.
25.2 Connecting to a server

The function `open-tcp-stream` connects to a server and returns a stream (of type `socket-stream`). The stream is then used for communication through the socket using the standard Lisp I/O functions.

`connect-to-tcp-server` can also be used, especially if you want to subclass `socket-stream`.

25.2.1 Examples

For examples illustrating simple write and read on a socket, see these files in `lib/7-1-0-0/examples`

- `capi/applications/chat.lisp`
- `capi/applications/chat-client.lisp`

25.3 Specifying the target for connecting and binding a socket

In general, each socket is bound to a local socket address, and is communicating with some other socket which has its own socket address. The local binding may be done implicitly by the system, but in many cases (in general, when it is a service) it needs to be bound to specific socket address. When connecting to another socket, or sending using UDP socket, the socket address of the other side is needed.

The socket address is always specified by a `hostspec` and `service`. `hostspec` is also referred to as "address" or "hostname" or "host", and the `service` is sometimes referred to as "port". In particular, the local `hostspec` and local `service` are called `local-address` and `local-port`.

`hostspec` specifies an IP address. It can be one of:

- A string naming the host, for example "www.google.com".
  
  Such a string is looked up by the system to find the actual IP address.

- A string providing the IP address in standard format.
  
  Example: "204.71.177.5" (IPv4)
25.4 Information about IP addresses

Example: "2001:500:2f::f" (IPv6).

An integer specifying IPv4 address in network order.

Example: #XCC47B14B.

An ipv6-address object.

The functions string-ip-address and ip-address-string convert between strings that specify addresses and integers or ipv6-address objects. If you need to find the actual address from a string giving the host name, you need to look it up using get-host-entry. Normally you do not need to, because all the interface functions do it implicitly.

service specifies the port number to use. It can be either an integer, which explicitly specifies the port number, or a string, which is either a sequence of decimal digits specifying the port number or a port name that is looked up to find the port number. For example, for http connections the port number is 80. The function get-service-entry can be used to convert between port numbers and names.

When connecting a socket (for example by open-tcp-stream), hostspec and service are required arguments. When binding (for example start-up-server), hostspec (which is normally passed by the keyword argument local-address) can be nil, which means use the local host and allow any connections. service (which is normally passed by the keyword argument local-port) can be specified as 0 or nil, both values meaning that the operating system will select some appropriate port number.

If you have a socket-stream or a socket, you can find what socket address it is bound to by socket-stream-address or get-socket-address, and if it is connected, you can find what address it is connected to by socket-stream-peer-address or get-socket-peer-address.

25.4 Information about IP addresses

You can use the function get-host-entry to find the address of a domain name or the domain name of an address. It can also used to find multiple addresses and aliases.
You can use `get-socket-address`, `get-socket-peer-address` and `socket-stream-address` and `socket-stream-peer-address` to find the IP address of opened sockets.

You can use `get-default-local-ipv6-address` to find the local default IPv6 address.

You can use `get-ip-default-zone-id` to find the local default zone ID.

### 25.5 Waiting on a socket stream

The function `wait-for-input-streams` and `wait-for-input-streams-returning-first` are a convenient interface for waiting for input from socket streams. The standard I/O functions (`cl:read`, `cl:read-char` and so on) can also wait properly. You can also use `process-wait` and similar functions with `cl:listen` in the `wait-function`, but you will need to use `with-noticed-socket-stream`.

### 25.6 Special considerations

The host machine must be configured properly to handle IPv6 for the LispWorks interface to work with IPv6.

It is likely that all new machines can use IPv6.

#### 25.6.1 IPv6 on Mac OS X

IPv6 addresses work properly on Mac OS X 10.6.

#### 25.6.2 IPv6 on Windows XP

By default IPv6 addresses do not work on Microsoft Windows XP. To make it work on Windows XP, install the interface by executing this command in a console, as an administrator user:

```
netsh interface ipv6 install
```

This should not be needed on later versions of Microsoft Windows. Search for `netsh` on `technet.microsoft.com` for more information.

**Note:** LispWorks 7.0 and later versions do not support Windows XP.
25.7 Asynchronous I/O

The Asynchronous I/O API allows you to perform I/O operations that invoke a callback when they are complete, rather than synchronously calling a function that returns a value (like `clojure.core/read-string`). This allows many operations to run in a single thread. When using this API, you have to hold all of the application's state in data structures so that the callback can determine how to proceed.

There are two parts to the API:

- the Wait-State-Collection API controls the overall progress of I/O.
- the Async-I/O-State API deals with individual I/O channels.

25.7.1 The wait-state-collection API

A `wait-state-collection` is an object that controls asynchronous I/O via an event loop. Each I/O channel is associated with a wait-state in the collection (see the “The Async-I/O-State API” on page 412 for how to add channels to a collection).


The function `loop-processing-wait-state-collection` simplifies processing I/O by repeatedly calling `wait-for-wait-state-collection` and `call-wait-state-collection`. It can be stopped by `wait-state-collection-stop-loop`. The function `create-and-run-wait-state-collection` makes a `wait-state-collection` and a process that runs it (using `loop-processing-wait-state-collection`). In many cases, `create-and-run-wait-state-collection` is the only function that you need to use.

To call a function in the process associated with a `wait-state-collection` you can use `apply-in-wait-state-collection-process` (but see also “Writing callbacks in Asynchronous I/O operations” on page 414).

For the `wait-state-collection` to actually do anything, it must have some "wait-states" associated with it. The primary way of associating "wait-states" with a `wait-state-collection` is to create an `async-io-state` associated
with it, see “The Async-I/O-State API” on page 412 below. The function
accept-tcp-connections-creating-async-io-states also creates an
associated "wait-state", which itself creates an async-io-state associated
with the wait-state-collection. Note that new async-io-states can be
added (and removed) dynamically to the wait-state-collection from any
process while it is working.

See

(example-edit-file "async-io/driver")

25.7.2 The Async-I/O-State API

The Async-I/O-State API contains functions to create and close various kinds
of asynchronous I/O channels and perform input and output operations on
them. Currently "I/O channel" means a socket or a socket-stream.

Each channel has an associated async-io-state object, which is used to
retain information about the channel between calls to the input and output
functions. You can store your own information using the async-io-state-
user-info accessor and give the object a name for debugging purposes using
the async-io-state-name accessor.

An async-io-state is created by any of these functions:

create-async-io-state
    Takes a socket (an integer) or a socket-stream and
    allows I/O on the socket.

create-async-io-state-and-connected-tcp-socket
    Takes a socket address to connect to, creates a TCP
    socket and connects it, and allows I/O on it.

accept-tcp-connections-creating-async-io-states
    Takes a service and creates a listening socket that
    accepts connection and create states which allow I/O
    on the accepted connections.

create-async-io-state-and-udp-socket
    Creates a UDP socket and allows I/O on it.
create-async-io-state-and-connected-udp-socket

Takes a socket address, creates a UDP socket and connect it, and allows I/O on it.

Once an async-io-state is created for an object, the object itself should not be used directly for I/O in the same direction (read or write). The async-io-state can then be made active by one of async-io-state-read-buffer, async-io-state-write-buffer, async-io-state-read-with-checking, async-io-state-receive-message, async-io-state-send-message and async-io-state-send-message-to-address.

Each async-io-state is associated with a wait-state-collection when it is created. For the async-io-state to be active, the wait-state-collection must be active, which means there must be a process calling wait-for-wait-state-collection and call-wait-state-collection, possibly via loop-processing-wait-state-collection.

The functions async-io-state-read-buffer and async-io-state-write-buffer create an I/O operation that reads or writes a fixed amount of data in a buffer. The operation finishes when the callback is called, or when when an abort-callback is called (after being set up by async-io-state-abort).

The function async-io-state-read-with-checking creates an input operation that periodically invokes a callback to determine whether enough data has been received, by examining the internal buffer. You can call async-io-state-discard to indicate that part of the internal buffer has been processed (for example parsed and converted to some data structure). The operation finishes when async-io-state-finish is called inside the callback, or when an abort-callback is called (after being set up by async-io-state-abort).

The function async-io-state-receive-message creates an input operation that receives a message (using recv or recvfrom). The functions async-io-state-send-message and async-io-state-send-message-to-address create an I/O operation that sends a message (using send or sendto). These three functions are intended to be used with states created with UDP sockets.

While an input operation is ongoing, you cannot start another input operation with the same direction. While a write operation is ongoing, whether you can start another write operation depends on the keyword argument :queue-output which is used when the async-io-state is created. If queue-input was
nil (the default for TCP), then you cannot start another write operation while one is ongoing. If queue-output was supplied as non-nil (the default for UDP), you can start another write operation, and the operation gets queued and actually starts after all previously queued operations have finished.

When you no longer need the async-io-state you must close it by close-async-io-state. Normally, that would close the object of the async-io-state too. close-async-io-state can be told to leave the object alive, so you can do further I/O with it. However, if you have read using async-io-state, it may have buffered data which you will need to deal with by async-io-state-buffered-data-length and async-io-state-get-buffered-data (unless you can just ignore it).

An async-io-state can have a name, to help identifying it, mainly for debugging. The default names that different functions give help to identify the kind of object that the state has.

See

(example-edit-file "async-io/multiplication-table")
(example-edit-file "async-io/print-connection-delay")
(example-edit-file "async-io/udp")

25.7.3 Writing callbacks in Asynchronous I/O operations

All of the Asynchronous I/O operations take a callback, which is called when the operation finished. The callbacks are called inside the same process that processes the wait-state-collection (specifically, the process that called call-wait-state-collection, potentially via loop-processing-wait-state-collection). That means that until the callback returns, no further processing happens on the wait-state-collection, and hence on any of the other async-io-states that are associated with it. Therefore callbacks need to be reasonably fast and not hang.

In general, the callbacks should be creating the next I/O operations, to ensure that that operations on each state are sequential (see “Asynchronous I/O and multiprocessing” on page 415). If this is a reasonably simple operation you just do it, but if the data for the next operation make take a long time to prepare you probably want to avoid doing it in the context of the callback. Things
that may cause it to take a long time include heavy computation or access to external resources that may cause delays.

A general solution is to send the work to another process, which will do the work and on completion will do the next I/O operation by calling the read/write \texttt{async-io-state} function.

Another possible solution is to perform operations that can be fast using one \texttt{wait-state-collection}, and perform slow operations on (an)other \texttt{wait-state-collection(s)}. This way a slow callback will only impede other slow callbacks. For example you may be accepting connections on the "fast" \texttt{wait-state-collection}, but communicate with the accepted connection on a slow \texttt{wait-state-collection} (pass \texttt{:create-state nil} to \texttt{accept-tcp-connections-creating-async-io-states}, and in the callback use \texttt{create-async-io-state} with another \texttt{wait-state-collection}). You may also decide to do the communication using streams and synchronous I/O (pass \texttt{:create-state nil} and in the \texttt{callback} use \texttt{(make-instance 'socket-stream ...) and send the result to another process).}

\subsection*{25.7.4 Asynchronous I/O and multiprocessing}

Processing of the \texttt{wait-state-collection} is not thread-safe, and for each collection there must be only one process at any one time calling any of these functions:
\begin{itemize}
  \item \texttt{loop-processing-wait-state-collection}
  \item \texttt{call-wait-state-collection}
  \item \texttt{wait-for-wait-state-collection}
  \item \texttt{close-wait-state-collection}
\end{itemize}

\texttt{wait-state-collection-stop-loop} is thread-safe, and can be called on any thread at any time.

Adding and removing states to/from the collection is thread-safe with respect to the collection, which means that the creation functions like \texttt{create-async-io-state} can be called in parallel with any function that access the same collection, including themselves and the processing functions above. The same applies to functions that remove the state from the collection (\texttt{close-async-}...
io-state), though these are not thread-safe with respect to the state (see below).

Note that the functions that create states use other resources which may have their own limitations. Most notably, local ports can be used only once at any time with the same protocol and family, so if you try to bind to a specific local port (by passing local-port to any of the functions or non-zero service in accept-tcp-connections-creating-async-io-states), you have to make sure that you do not do it with a port that is currently in use. (Note that accept-tcp-connections-creating-async-io-states may try several times).

The functions that actually do the I/O are not thread-safe with respect to the state argument, but are thread-safe with respect to the collection that the state is associated with. That means that they can be called in parallel to any function that accesses the collection that the state is associated with, but cannot be called in parallel to another function that tries to do I/O on the same state and direction. Moreover, the read functions cannot be called while there is an ongoing read operation, and the write function can be called while another write operation is ongoing only if queue-output is non-nil when creating the state. The function close-async-io-state also cannot be called in parallel to any of the I/O functions.

Explicitly:


If queue-output was nil when the state was created (TCP default), the writing functions async-io-state-write-buffer, async-io-state-send-message, and async-io-state-send-message-to-address must not be called on the same state in parallel to any of themselves, or in the period between any call to any of themselves and the call to the callback, or abort-callback. If queue-output was non-nil when the state was created (UDP default), the writing functions can be called in parallel.
close-async-io-state must not be called on the same state in parallel to any of the reading or writing functions, or between a call to any of them at the end of their operation (the callback, async-io-state-finish, or the abort-callback).

The reading and writing functions are mutually thread-safe, that is any of the reading functions can be called in parallel to any of the writing functions.

The functions async-io-state-abort and async-io-state-abort-and-close are thread-safe, and be called at any time in parallel to any function.

async-io-state-get-buffered-data is not thread-safe, and must not be called in parallel to any other function that may modify the state.

async-io-state-finish and async-io-state-discard are not thread-safe, but can only be called inside the callback of async-io-state-read-with-checking, which will be always in the same process. The accessors of async-io-state are thread-safe.

In general, it is intended that you will cope with these thread-safe restrictions of I/O functions by calling them from the callbacks of the previous I/O operation, thus guaranteeing that the previous I/O operation finished. For example, if you need to write several buffers to a socket, you can call async-io-state-write-buffer with the first buffer, and with a callback that calls async-io-state-write-buffer with the next buffer. A natural place to put the information where to get the next buffer is the user-info of the async-io-state, which can be accessed using async-io-state-user-info. For example, assume you have an async-io-state, a list of buffers to send, and also on completion you want to call a function finished on some object:

```
(defun my-send-buffers (state buffers object)
  (setf (async-io-state-user-info state)
        (cons buffers object))
  (my-state-send-next-buffer state))

(defun my-state-send-next-buffer (state)
  (let ((info (async-io-state-user-info state)))
    (if-let (buffer (pop (car info)))
       (async-io-state-write-buffer state buffer
         #'(lambda (state buffer length)
            (declare (ignore buffer length))
            (my-state-send-next-buffer state)))
       (finished (cdr info))))
```
In a real application the *user-info* is likely to be a more complex object.

If you make the *state* with *queue-output t*, you can simply write all the buffers in one go:

```lisp
(defun my-send-buffers (state buffers object)
  (setf (async-io-state-user-info state) object)
  (loop for cons on buffers
    do
    (async-io-state-write-buffer
     state (car cons)
     :callback
     (if (cdr cons) ; if there are more buffers
       #'true ; do nothing
       #'(lambda (state buffer length)
          (declare (ignore buffer length))
          (finished
           (async-io-state-user-info state))))))
```

### 25.8 Using SSL

The SSL interface allows you to use Secure Socket Layer (SSL) with Lisp objects of type *socket-stream* and *async-io-state*.

The interface is based on the OpenSSL code, and most of it is simply an FLI interface to OpenSSL functions. The main LispWorks specific code is the way OpenSSL is integrated with *socket-stream* and *async-io-state*.

The SSL interface is part of the "comm" module, so to load it you evaluate

```lisp
(require "comm")
```

**Note:** In this section we assume that the current package uses the *comm* package. That is, *comm* package symbols may not be qualified explicitly.

#### 25.8.1 Obtaining and installing the OpenSSL library

At the time of writing, OpenSSL is available as shown in Table 25.1:

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Availability of OpenSSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>Installed by default on most 32-bit and 64-bit distributions</td>
</tr>
</tbody>
</table>
25.8 Using SSL

25.8.1 Installing the OpenSSL library on Solaris

After installing (with pkgadd) you need to put the shared libraries `libcrypto.so` and `libssl.o` on the loader path. By default these are installed in `/usr/local/ssl/lib`.

To add the libraries to the loader path, either

- Add `/usr/local/ssl/lib` to the environment variable `LD_LIBRARY_PATH`, or
- Create links from `/usr/lib`.

### Table 25.1 OpenSSL availability

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Availability of OpenSSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>32-bit and 64-bit libraries are available at <a href="http://www.slproweb.com/products/Win32OpenSSL.html">www.slproweb.com/products/Win32OpenSSL.html</a></td>
</tr>
<tr>
<td>Mac OS X</td>
<td>32-bit and 64-bit libraries are installed by default.</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>Installed by default and available via ports or pkg.</td>
</tr>
<tr>
<td>AIX</td>
<td>Installed by default</td>
</tr>
<tr>
<td>x86/x64 Solaris</td>
<td>Installed by default</td>
</tr>
<tr>
<td>SPARC Solaris</td>
<td>Installed by default on Solaris 10. For other versions, see the freeware from Sun at <a href="http://sunfreeware.com">sunfreeware.com</a> for both 32-bit and 64-bit.</td>
</tr>
</tbody>
</table>
25.8.1.2 How LispWorks locates the OpenSSL libraries

Since OpenSSL is not a standard on all machines yet, the location of the library or libraries varies. By default, ensure-ssl loads libraries as shown in Table 25.2, page 420.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>-1ssl</td>
</tr>
<tr>
<td>Windows</td>
<td>libeay32.dll</td>
</tr>
<tr>
<td></td>
<td>libssl32.dll</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>-1crypto -1ssl</td>
</tr>
<tr>
<td>Solaris</td>
<td>-1ssl</td>
</tr>
<tr>
<td>AIX</td>
<td>-1ssl</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>-1ssl</td>
</tr>
<tr>
<td>Others</td>
<td>nil</td>
</tr>
</tbody>
</table>

On machines where the path is unknown or is incorrect, you must set the path by calling set-ssl-library-path, or by passing the path as the library-path argument to ensure-ssl. In particular, if you want to use the Windows OpenSSL 1.1 libraries from the page that is mentioned in the table “OpenSSL availability” on page 418, then the library names have changed and you will have to set the path. With the download from that page from September 2017, the names need to be:

```
64-bit           ("libssl-1_1-x64.dll"
                  "libcrypto-1_1-x64.dll")
32-bit           ("libssl-1_1.dll"
                  "libcrypto-1_1.dll")
```

25.8.2 Creating a stream with SSL

There are three ways to make a socket-stream with SSL processing:
25.8 Using SSL

- Call (make-instance 'socket-stream :ssl-ctx ...)
- Call open-tcp-stream with the :ssl-ctx keyword.
- Call attach-ssl on a socket-stream.

These calls implicitly load the OpenSSL library and seed the Pseudo Random Number Generator (PRNG).

For example:

(open-tcp-stream some-url 443 :ssl-ctx t)

25.8.3 Using Asynchronous I/O with SSL

There are three ways to make an async-io-state with SSL processing:

- Call accept-tcp-connections-creating-async-io-states with the :ssl-ctx keyword.
- Call async-io-state-attach-ssl on an async-io-state.

These calls implicitly load the OpenSSL library and seed the Pseudo Random Number Generator (PRNG).

25.8.4 Keyword arguments for use with SSL

The keyword arguments :ssl-ctx, :ssl-side, :ctx-configure-callback, :ssl-configure-callback and :handshake-timeout can be be passed to create and configure socket streams and async-io-states with SSL processing.

The various interface calls for creating and configuring SSL streams and
async-io-states accept these keyword arguments as shown in Table 25.3, page 422.

Table 25.3 SSL configuration keywords

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>socket-stream make-instance</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>open-tcp-stream</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>attach-ssl</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>make-ssl-ctx</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>accept-tcp-connections-creating-async-io-states</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>create-async-io-state-and-connected-tcp-socket</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>async-io-state-attach-ssl</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

(make-instance 'socket-stream ...) and open-tcp-stream, when ssl-ctx is non-nil, call attach-ssl and pass it all the arguments. accept-tcp-connections-creating-async-io-states and create-async-io-state-and-connected-tcp-socket when ssl-ctx is non-nil attach the ssl similar to the way async-io-state-attach-ssl does.
:ssl-ctx specifies that SSL should be used, and also specifies the SSL_CTX object to use. See the OpenSSL manual entry for SSL_CTX_new for details of making a SSL_CTX. The value of ssl-ctx can be:

A symbol Together with ssl-side, this symbol specifies which protocol to use. ssl-ctx can be one of:

1) t or :default, meaning use the default. Currently this is the same as :v23.

2) One of :v2, :v3, :v23 or :tls-v1. These are mapped to the SSLv2_*, SSLv3_*, SSLv23_*, TLSv1_* methods.

LispWorks makes a new SSL_CTX object and uses it and frees it when the stream or state is closed. The interface calls also make an SSL object, uses it and frees it when the stream or state is closed.

A foreign pointer of type ssl-ctx-pointer
This corresponds to the C type SSL_CTX*. This is used and is not freed when the stream is closed. The interface calls also make an SSL object, use it and free it when the stream is closed. The foreign pointer maybe a result of a call to make-ssl-ctx, but it can also be a result of your code, provided that it points to a valid SSL_CTX and has the type ssl-ctx-pointer.

A foreign pointer of type ssl-pointer
This corresponds to the C type SSL*. This specifies the SSL to use in the interface calls. This may be a result of a call to ssl-new, but can also be a result of your code, provided that it points to a valid SSL object and has the type ssl-ctx-pointer. The SSL is used and is not freed when the stream is closed.

When you pass a ssl-ctx-pointer or a ssl-pointer foreign pointer, these must have already been set up correctly and you are responsible for freeing them when they are no longer required.
:ssl-side specifies which side the stream is. The value ssl-side can be one of :client, :server or :both. open-tcp-stream and create-async-io-state-and-connected-tcp-socket do not take this keyword and always use :client. For the other calls this argument defaults to :server. The value of ssl-side is used in three cases:

- When a new SSL_CTX object is created, it is used to select the method:
  :client => *_client_method
  :server => *_server_method
  :both => *_method

- When ssl-ctx is of type ssl-ctx-pointer, it checks that the side of ssl-ctx and ssl-side are not conflicting. If one is :client and the other is :server, they conflict and an error is signaled.

- When a new SSL object is created, when ssl-side is either :client or :server, LispWorks calls ssl-set-connect-state or ssl-set-accept-state respectively.

When a new SSL object is created, ssl-side is :client and handshake-timeout is greater than 0, a handshake is performed immediately.

If ssl-ctx is of type ssl-pointer then ssl-side is ignored.

:ctx-configure-callback specifies a callback, a function which takes a foreign pointer of type ssl-ctx-pointer. This is called immediately after a new SSL_CTX is created. If the value of ssl-ctx is not a symbol, ctx-configure-callback is ignored.

:ssl-configure-callback specifies a callback, a function which takes a foreign pointer of type ssl-pointer. This is called immediately after a new SSL is created. If the value of ssl-ctx is a ssl-pointer, ssl-configure-callback is ignored.

When a handshake is performed immediately (ssl-side is :client and ssl-ctx is not a ssl-ctx-pointer), handshake-timeout specifies the time in seconds to wait for the handshake to complete. If handshake-timeout is nil (the default) then it waits forever. If the handshake fails or times out, it is an error situation: for the synchronous interface (stream) an error is signaled, and for the asynchronous interface the callback is called with an error indicator (see the spe-
specific functions for details). Note that `handshake-timeout 0` (or negative) prevents
the handshake.

In typical usage, you will create few `ssl-ctx-pointer` objects (maybe only
one), configure them as appropriate for your application and the machine that
it runs on, and then use one of these as `ssl-ctx` in all of your calls. If some con-
nections need special configuration, you will use `ssl-configure-callback` to con-
figure the `ssl-pointer` of this connection. Sometimes when you open a
connection as a client it may be sufficient to pass a symbol for `ssl-ctx`. Passing
an `ssl-pointer` as `ssl-ctx` is for special cases.

### 25.8.5 Attaching SSL to an existing socket

You can attach SSL to an existing `socket-stream` by calling `attach-ssl` on
the stream. The `socket-stream` SSL keyword arguments are processed by
`attach-ssl` as described in “Keyword arguments for use with SSL” on page
421.

Detach SSL from a `socket-stream` and shut down the SSL with `detach-ssl`.
For full descriptions see `attach-ssl`, page 554 and `detach-ssl`, page 575.

You can attach SSL to an existing `async-io-state` by calling `async-io-
state-attach-ssl` on the state, and detach it using `async-io-state-
detach-ssl`.

Notes:

After an object (stream or state) has been detached, you can attach SSL to it
again.

Detaching frees any automatically generated SSL objects in the same way that
closing a stream or state does.

The SSL objects are attached to the `socket-stream` or `async-io-state`,
rather than to the socket. Therefore if you want to move a socket to another
object then you need to attach it again.

For example, if you have attached SSL to an `async-io-state` and then want
to change to synchronous communication, you need to close the `async-io-
state` by `close-async-io-state` with `keep-alive` true (effectively detach the
SSL), and then call make-instance with socket-stream with the socket plus SSL-CTX and any other necessary arguments.

To move the other way, from synchronous to asynchronous, use replace-socket-stream-socket with socket nil to disconnect the socket from the stream (which effectively calls detach-ssl), call create-async-io-state with the socket, and then call async-io-state-attach-ssl on the new async-io-state.

### 25.8.6 OpenSSL interface

The configuration interface contains mostly FLI function definitions that map directly to OpenSSL calls. See below for a list of those provided.

There are also some functions to make common cases simpler. These are read-dhparams, pem-read, set-ssl-ctx-options, set-ssl-ctx-password-callback, and set-ssl-ctx-dh.

#### 25.8.6.1 OpenSSL constants

The Lisp constants SSL_FILETYPE_ASN1 and SSL_FILETYPE_PEM representing file types are provided.

#### 25.8.6.2 Naming conventions for direct OpenSSL calls

This section describes the mapping between OpenSSL function names and the corresponding Lisp names.

#### 25.8.6.3 Mapping C names to Lisp names

For functions that map directly to OpenSSL calls, the convention is to create the LISP name from the C name by replacing underscores by hyphens.

#### 25.8.6.4 Mapping Lisp names to C names

To find the C name from the LISP function name:

1. the hyphens need to be replaced by underscores, and
2. the initial SSL or SSL_CTX has to be in uppercase, and
3. the rest has to be lowercase, except that
4. the following phrases are cased specially, like this: "RSAPrivateKey", "DSH", "ASN1", "CA", "PrivateKey"

25.8.7 Direct calls to OpenSSL

The following functions map directly to the OpenSSL functions. Check the OpenSSL documentation for details.

Where an OpenSSL function takes an SSL* or SSL_CTX*, the Lisp function’s argument must be a foreign pointer of type ssl-pointer, ssl-ctx-pointer or ssl-cipher-pointer. Where an OpenSSL function takes a char* or int, the Lisp function’s argument must be a string or integer. Where an OpenSSL function takes other kinds of pointers, the Lisp function’s argument must be a foreign pointer. The return values are integers or foreign pointers unless stated otherwise.

If an error occurs in one of these functions, an error code is returned. They do not signal any Common Lisp conditions and so you should check the return value carefully.

Table 25.4 Direct calls to OpenSSL

<table>
<thead>
<tr>
<th>Lisp function</th>
<th>Return values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ssl-add-client-ca</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-cipher-get-bits</code></td>
<td>First value is number of bits the cipher actually uses.</td>
</tr>
<tr>
<td></td>
<td>Second value is number of bits the algorithm of the cipher can use (which</td>
</tr>
<tr>
<td></td>
<td>may be higher).</td>
</tr>
<tr>
<td><code>ssl-cipher-get-name</code></td>
<td>string.</td>
</tr>
<tr>
<td></td>
<td>e.g. &quot;DHE-RSA-AES256-SHA&quot;</td>
</tr>
<tr>
<td><code>ssl-cipher-get-version</code></td>
<td>string.</td>
</tr>
<tr>
<td></td>
<td>e.g. &quot;TLSv1/SSLv3&quot;</td>
</tr>
<tr>
<td><code>ssl-clear-num-renegotiations</code></td>
<td></td>
</tr>
</tbody>
</table>
Table 25.4  Direct calls to OpenSSL

<table>
<thead>
<tr>
<th>Lisp function</th>
<th>Return values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssl-ctrl</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-add-client-ca</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-add-extra-chain-cert</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-ctrl</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-get-max-cert-list</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-get-mode</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-get-options</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-get-read-ahead</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-get-verify-mode</td>
<td>integer</td>
</tr>
<tr>
<td>ssl-ctx-load-verify-locations</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-need-tmp-rsa</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-sess-set-cache-size</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-sess-get-cache-size</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-sess-set-cache-mode</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-sess-get-cache-mode</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-set-client-ca-list</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-set-max-cert-list</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-set-mode</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-set-options</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-set-read-ahead</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-set-tmp-rsa</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-set-tmp-dh</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-use-certificate-chain-file</td>
<td></td>
</tr>
</tbody>
</table>
Table 25.4 Direct calls to OpenSSL

<table>
<thead>
<tr>
<th>Lisp function</th>
<th>Return values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ssl-ctx-use-certificate-file</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-ctx-use-privatekey-file</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-ctx-use-rsaprivatekey-file</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-get-current-cipher</code></td>
<td><code>ssl-cipher-pointer</code></td>
</tr>
<tr>
<td></td>
<td>Can be a null pointer.</td>
</tr>
<tr>
<td><code>ssl-get-max-cert-list</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-get-mode</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-get-options</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-get-verify-mode</code></td>
<td><code>integer</code></td>
</tr>
<tr>
<td><code>ssl-get-version</code></td>
<td><code>string</code></td>
</tr>
<tr>
<td></td>
<td>&quot;TLSv1&quot;, &quot;SSLv2&quot; or &quot;SSLv3&quot;</td>
</tr>
<tr>
<td><code>ssl-load-client-ca-file</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-need-tmp-rsa</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-num-renegotiations</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-session-reused</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-set-accept-state</code></td>
<td><code>None</code></td>
</tr>
<tr>
<td><code>ssl-set-client-ca-list</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-set-connect-state</code></td>
<td><code>None</code></td>
</tr>
<tr>
<td><code>ssl-set-max-cert-list</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-set-mode</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-set-options</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-set-tmp-rsa</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-set-tmp-dh</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-total-renegotiations</code></td>
<td></td>
</tr>
</tbody>
</table>
If you need OpenSSL functionality that is not provided here, you can define your own foreign functions via the LispWorks Foreign Language Interface. If you do this, an important point to note is that on Microsoft Windows, the :calling-convention must be :cdecl (it defaults to :stdcall). If using OpenSSL suddenly causes mysterious crashes, the calling-convention in your foreign function definitions is the first thing to check.

### 25.8.8 Using SSL objects directly

The C objects SSL and SSL_CTX are represented in LispWorks by foreign pointers with type `ssl-pointer` and `ssl-ctx-pointer`, which correspond to the C types SSL* and SSL_CTX*. These foreign types should be used for any foreign function that takes or returns these C types, and must be used when passing a foreign pointer as the value of the :ssl-ctx argument.

Making SSL objects is a way of getting access to them to perform configuration, but, especially in the case of the SSL_CTX, it is a useful way to avoid repeated calls to the configuration routines which may be time consuming. For example, if we have defined a function `configure-a-ctx`, and we want to read once every 60 seconds from some URL, we can write:

```lisp
(loop (with-open-stream
    (str (comm:open-tcp-stream some-url 443 :ssl-ctx t
        :ctx-configure-callback
        'configure-a-ctx))
    (read-something str))
    (sleep 60))
```

This will cause `configure-a-ctx` to be called each time. If it is expensive, we can call it only once by changing the code to:

---

Table 25.4 Direct calls to OpenSSL

<table>
<thead>
<tr>
<th>Lisp function</th>
<th>Return values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ssl-use-certificate-file</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-use-rsaprivatekey-file</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-use-privatekey-file</code></td>
<td></td>
</tr>
</tbody>
</table>

If you need OpenSSL functionality that is not provided here, you can define your own foreign functions via the LispWorks Foreign Language Interface. If you do this, an important point to note is that on Microsoft Windows, the :calling-convention must be :cdecl (it defaults to :stdcall). If using OpenSSL suddenly causes mysterious crashes, the calling-convention in your foreign function definitions is the first thing to check.

### 25.8.8 Using SSL objects directly

The C objects SSL and SSL_CTX are represented in LispWorks by foreign pointers with type `ssl-pointer` and `ssl-ctx-pointer`, which correspond to the C types SSL* and SSL_CTX*. These foreign types should be used for any foreign function that takes or returns these C types, and must be used when passing a foreign pointer as the value of the :ssl-ctx argument.

Making SSL objects is a way of getting access to them to perform configuration, but, especially in the case of the SSL_CTX, it is a useful way to avoid repeated calls to the configuration routines which may be time consuming. For example, if we have defined a function `configure-a-ctx`, and we want to read once every 60 seconds from some URL, we can write:

```lisp
(loop (with-open-stream
    (str (comm:open-tcp-stream some-url 443 :ssl-ctx t
        :ctx-configure-callback
        'configure-a-ctx))
    (read-something str))
    (sleep 60))
```

This will cause `configure-a-ctx` to be called each time. If it is expensive, we can call it only once by changing the code to:
(let ((ctx (comm:make-ssl-ctx :ssl-side :client)))
  (configure-a-ctx ctx)
  (loop (with-open-stream
    (str (comm:open-tcp-stream some-url 443 :ssl-ctx ctx))
    (read-something str))
    (sleep 60))
  (comm:destroy-ssl-ctx ctx))

The SSL objects could be made either by make-ssl-ctx or ssl-new or by user
code that calls the C functions SSL_CTX_new and SSL_new. destroy-ssl-
ctx frees the SSL_CTX object. To free an SSL object you would call destroy-
ssl. See the manual entries for full descriptions of these functions.

Alternatively, the SSL objects can be obtained from a socket-stream by call-
ing socket-stream-ssl or socket-stream-ctx and from an async-io-
state by calling async-io-state-ssl or async-io-state-ssl. You can
also find the ssl-side value that was passed to the interface call that created the
SSL objects by calling socket-stream-ssl-side or async-io-state-ssl-
side.

### 25.8.9 Initialization

All the functions that make a SSL_CTX first call ensure-ssl, so normally you
do not need to initialize the library. If your code makes a SSL_CTX itself (that
is, not by calling any of the LispWorks interface functions), it needs to initial-
ize the library first. Normally that should be done by an explicit call to
ensure-ssl, which loads the SSL library and calls SSL_library_init and
SSL_load_error_strings, and also does some LispWorks specific initializations.
If your code must do the initialization, ensure-ssl should still be called with
the argument :already-done t, which tells it that the library is already
loaded and initialized.

### 25.8.10 Errors in SSL

If there are errors inside SSL, LispWorks will signal an error of type ssl-con-
dition, which is a subclass of socket-error.

The condition can be one of the types ssl-x509-lookup, ssl-closed, ssl-
error and ssl-failure. See the manual pages for details of these condition
classes.
25.8.11 Examples of using the socket stream SSL interface

See the example files in:

(example-edit-file "ssl")

25.9 Socket streams with Java sockets and SSL on Android

Socket streams can now be implemented on top of Java Objects, instead of native sockets. The main purpose of this is to allow using SSL in LispWorks for Android Runtime, because OpenSSL is not available on Android. It may also be useful where you have a Java socket from some source and want to communicate through it using a Lisp stream.

The function `switch-open-tcp-stream-with-ssl-to-java` is called automatically before delivery for Android by `deliver-to-android-project`. That causes `open-tcp-stream`, when it is called with `ssl-ctx` non-nil, to use Java sockets instead of operating system sockets.

The function `open-tcp-stream-using-java` can be used to force using a Java socket.

You can also explicitly create a stream using Java sockets by passing a Java socket to `(make-instance 'comm:socket-stream ...)` or by setting the socket in an existing stream using `(setf comm:socket-stream-socket)` or `replace-socket-stream-socket`.

Socket streams with Java sockets are limited, mainly because `cl:listen` cannot be used reliably with them. Specifically, when `cl:listen` returns `t` it is guaranteed that reading will not hang, but when `cl:listen` returns `nil` it does not mean that there is nothing to read. They also do not have a zero timeout: the shortest timeout is 1 millisecond. That means that it is impossible to check whether reading from the stream will hang. The best that you can do is to set the `read-timeout` to a short time, and then try to read.

There is also no write timeout.

The Asynchronous I/O interface and the server side (`start-up-server`) do not work at the moment with Java sockets. If you want to create a service with Java sockets, you will need to implement the listening part using Java methods. Once a socket is accepted, you can pass it to `(make-instance 'comm:socket-stream ...)` to do the actual communication.
25.9 Socket streams with Java sockets and SSL on Android

When using Java sockets, the SSL configuration arguments `ctx-configure-callback` and `ssl-configure-callback`, as well as the `write-timeout` and `ipv6`, are ignored. The `ssl-ctx` is ignored when passed to `cl:make-instance`, and when passed to `open-tcp-stream` or `open-tcp-stream-using-java` it is interpreted as a boolean, specifying whether to use SSL or not. The only way to configure the socket, and more importantly the SSL settings, is by passing a socket factory (a Java object of class `javax.net.SocketFactory`) to `open-tcp-stream-using-java`. The application needs to set up and configure this factory using Java methods. By default, `open-tcp-stream` and `open-tcp-stream-using-java` use the default factory (which they get by the method "getDefault" on `javax.net.SocketFactory` or `javax.net.ssl.SSLSocketFactory`). Thus configuring the default factories affects what they do.

`cl:listen` is unreliable because the only way to check whether there is input on a Java socket is to use the Java method "available" on the input stream of the Java socket (that is, the result of the method "getInputStream"). The "available" method is documented as unreliable, and experimentally it is indeed unreliable on SSL sockets (on plain sockets it seems to work properly). If you know that in the implementation that you use the method "available" on an input stream of a socket is reliable, then you can trust `cl:listen` on socket streams with Java sockets.

Using Java sockets requires the LispWorks Java interface running Java Virtual Machine (JVM). On Android there is always a running JVM. On other architectures the JVM must be initialized by `init-java-interface`. To load the LispWorks Java interface, do

```lisp
(require "java-interface")
```

When using Java sockets and SSL, the default behavior is to verify the hostname (not done on the ordinary sockets). To do that it relies on classes from `httpclient` from `apache.org`, so `httpclient` must be in the class path for using Java sockets with SSL. `httpclient` is always available on Android. See `open-tcp-stream-using-java` for details of the verification process.

25.9.1 Android-specific points

On Android, the OpenSSL library is not available, so if the module "comm" was loaded, `deliver-to-android-project` switches to using Java sockets
for SSL streams. These streams have problems with \texttt{cl:listen}, discussed above. In principle, if you can find OpenSSL library for Android you can switch it back by calling \texttt{switch-open-tcp-stream-with-ssl-to-java} with \texttt{nil}, and use SSL in the usual way. You need to use \texttt{set-ssl-library-path} to tell the system where to find the library.

Android does not allow doing socket operations on the GUI threads (since Honeycomb SDK), and doing such operations would give a \texttt{java-exception} error with exception \texttt{NetworkOnMainThreadException}. That applies to \texttt{socket-stream} where the socket is a Java socket. However, it is always a bad idea to do socket operations on the GUI thread, so you should not do socket operations in the GUI thread with ordinary sockets either.
Internationalization: characters, strings and encodings

26.1 Introduction

LispWorks uses Unicode internally in its representation of character objects. All Unicode characters can be represented in strings, though 8-bit and 16-bit strings are also provided for efficiency when characters beyond the Latin-1 range (up to code #xff) or the BMP (Basic Multilingual Plane, up to code #xffff) respectively are not needed.

Character and string data can be input and output in various encodings (external formats).

26.2 Unicode support

Character implementation in LispWorks covers the full range of the Unicode standard.

cl:char-code-limit is #x110000, which covers exactly the Unicode range. The surrogate code points (codes #xd800 to #xdfff) are illegal as character codes.

cl:code-char accepts integers from 0 below cl:char-code-limit. Other values cause an error. For codes in the surrogate range it returns nil. Reading characters from streams and converting characters from foreign strings can generate characters in all the range (depending on the external-format used),
and can never generate character objects corresponding to surrogate code points.

`text-string` and `simple-text-string` take 32 bits per character and can store the full range of Unicode characters.

`simple-char` is now a synonym for `cl:character`, and is deprecated.

16-bit characters and 16-bit strings are implemented by types `bmp-char` and `bmp-string` and `simple-bmp-string` (BMP is Basic Multilingual Plane, the first plane (0 - `#x10000`) of Unicode). You may want to use `bmp-string` to minimize memory usage if you have an application with many 16-bit strings. That will work provided all the characters you ever use have codes less than `#x10000`. If all of the codes are below 256, you can use `base-string` instead.

**Note:** Character bits and font attributes are not supported. To deal with bits, use Gesture Spec objects (see `make-gesture-spec` and `coerce-to-gesture-spec`).

### 26.3 Character and String types

#### 26.3.1 Character types

LispWorks supports all the characters in the Unicode range `[0, #x10ffff]`, excluding the surrogate range `[#xd800, #xdbff]`. Note that character objects corresponding to surrogate code points may be produced by some APIs in LispWorks, but not by the interfaces that you should normally use to generate characters and strings in Common Lisp (that is `cl:code-char`, reading from a stream, converting from a foreign string, loading and storing from or to strings).

The following subtypes of character are defined:

- **base-char**: Characters with `cl:char-code` less than `base-char-code-limit (256).`
- **bmp-char**: Characters with `cl:char-code` less than `#x10000` (BMP stands for Basic Multilingual Plane in Unicode).
- **character**: All characters.
26.3.2 Compatibility notes

In LispWorks 6.1 and earlier versions, characters with codes up to \#x10000 are supported, and surrogate code points are allowed.

`bmp-char` was new in LispWorks 7.0, and matches the range of characters in LispWorks 6.1 and earlier versions, except that surrogate code points are no longer valid.

In LispWorks 6.1 and earlier versions there is `simple-char` which is now a synonym for `cl:character`. Using `cl:character` is preferable and portable.

In LispWorks 6.1 and earlier versions character bits attributes are supported, and also some characters represent keyboard gestures. These are no longer supported.

26.3.3 Character Syntax

All simple characters have names that consist of U+ followed by the code of the character in hexadecimal, for example \#\U+764F is `(code-char #x764F)`.

The hexadecimal number must be 4-6 characters, for example \#\U+a0 is illegal. Use \#\U+00a0 instead.

Additionally, Latin-1 characters have names derived from the ISO10646 name, for example:

```
  (char-name (code-char 190))
=>
  "Vulgar-Fraction-Three-Quarters"
```

Names are also provided for space characters:

```
  (name-char "Ideographic-Space")
=>
  \#\Ideographic-Space
```

Note that surrogate characters, that is the inclusive range [\#xd800, \#xdfff] are not acceptable, and trying to read such a character, for example \#\U+D835, produces an error.
26.3.4 Compatibility notes

In LispWorks 6.1 and earlier versions you can specify bits in character names. This is illegal in LispWorks 7.0 and later.

In LispWorks 6.1 and earlier versions character codes are limited to less than \#x10000, and surrogate code points are allowed.

26.3.5 String types

String types are supplied which are capable of holding each of the character types mentioned above. The following string types are defined:

- **base-string**: holds any base-char.
- **bmp-string**: holds any bmp-char.
- **text-string**: holds any cl:character (see “Character types” on page 436).

Compatibility note: bmp-string was new in 7.0. In LispWorks 6.1 and earlier versions there is augmented-string, this is now a synonym for text-string and is deprecated.

In LispWorks 6.1 and earlier versions, text-string could hold characters with codes less than \#x10000.

The types above include non-simple strings - those which are displaced, adjustable or with a fill-pointer.

The Common Lisp type string itself is dependent on the value of *default-character-element-type* according to the rules for string construction described in “String Construction” on page 441. For example:
The following types are subtypes of `simple-string`. Note that in the names of the string types, `simple` refers to the string object and does not mean that the string’s elements are `simple-char`.

- **simple-base-string**
  - holds any `base-char`.
- **simple-bmp-string**
  - holds any `bmp-char`.
- **simple-text-string**
  - holds any `cl:character`.

The Common Lisp type `simple-string` itself is dependent on the value of `*default-character-element-type*` according to the rules for string construction described in “String Construction” on page 441.

### 26.3.5.1 String types at run time

The type `string` (and hence `simple-string`) is defined by ANSI Common Lisp to be a union of all the character array types. This makes a call like

```
(coerce s 'simple-string)
```
ambiguous because it needs to select a concrete type (such as simple-base-string or simple-text-string).

When LispWorks is running with *default-character-element-type* set to base-char, it expects that you will want strings with element type base-char, so functions like coerce treat references to simple-string as if they were (simple-array base-char (*)).

If you call set-default-character-element-type with a larger character type, then simple-string becomes a union of the array types that are subtypes of that character type.

### 26.3.5.2 String types at compile time

The compiler always does type inferencing for simple-string as if *default-character-element-type* was set to character.

For example, when you declare something to be of type simple-string, the compiler will never treat it as simple-base-string. Therefore calls like

```
(schar (the simple-string x) 0)
```

will work whether `x` is a simple-base-string, simple-bmp-string or simple-text-string.

### 26.4 String accessors

`schar` works on any simple string object. However, for efficient string access when a simple string type is known, the following specialized accessors are provided:

```
sbchar for simple-base-string.
stchar for simple-text-string.
```

For simple-bmp-string there is no explicit accessor, but you can get the optimized access by declaring it as simple-bmp-string, and do the access using cl:schar.

You can also use declarations to optimize the access to simple-base-string and simple-text-string. In the case of simple-base-string, that means using only Common Lisp symbols, so it is fully portable.
26.5 String Construction

LispWorks constructs strings of a suitable type where sufficient information is available. Failing that, strings are constructed of type according to the value of *default-character-element-type*.

26.5.1 Default string construction

If the value of *default-character-element-type* is `base-char` then:

```lisp
(make-string 3)
```
returns a `simple-base-string` and

```lisp
(coerce sequence 'simple-string)
```
attempts to construct a `simple-base-string`. This will signal an error if any element of `sequence` is not a `base-char`.

If the value of *default-character-element-type* is `cl:character` then

```lisp
(make-string 3)
```
returns a `simple-text-string` and

```lisp
(coerce sequence 'simple-string)
```
attempts to construct a `simple-text-string`. This will signal an error if any element of `sequence` is not a `cl:character`.

Other string constructors also take their default from *default-character-element-type*. For instance, `with-output-to-string` and `make-string-output-stream` will construct a stream with element type determined by this variable and generate a string of the same element type.

Also the string reader will always construct a string of type determined by *default-character-element-type*, unless it sees a character of a larger type, in which case a suitable string is constructed. For example:

```lisp
CL-USER 1 > (set-default-character-element-type 'character)
CHARACTER

CL-USER 2 > (type-of "ABC")
SIMPLE-TEXT-STRING
```
Compatibility note: In LispWorks 6.0 and earlier versions, the string reader would not always obey *default-character-element-type*, due to a bug.

26.5.2 String construction with known type

The parameter *default-character-element-type* merely provides the default behavior. If enough information is supplied, then a string of suitable type is constructed. For instance, the form:

```
(make-string 3 :initial-element #
\Ideographic-Space
```

constructs a string of a type that can hold its elements, regardless of the value of *default-character-element-type*.

Likewise, format nil, princ-to-string, prin1-to-string and write-to-string will return a string whose element type can hold all characters that are written.

Functions that have a sequence type specifier as an argument, such as concatenate, use it as described in “String types” on page 438.

26.5.3 Controlling string construction

The initial value of *default-character-element-type* is base-char, to avoid programs that only require 8-bit strings needlessly creating larger string objects. If your application uses Unicode characters beyond the Latin-1 range (characters of type extended-char) then you should consider which of the following two approaches to use:

- Ensure that all strings which may hold extended-chars are constructed explicitly with the appropriate type. This is the conservative approach, allowing you to avoid allocation of 16-bit strings where these are not required. Note that you can use the specialized accessors such as stchar for strings of type simple-text-string.

- Change the default so that by default 16-bit strings are allocated. Do this by:

```
(set-default-character-element-type 'cl:character)
```

Bear in mind that this is a global setting which affects default string construction for the entire system. It could be called from a user interface, depending on whether the user needs to handle extended-chars.
Note: Do not attempt to bind or set directly the variable *default-character-element-type*. Instead, call `set-default-character-element-type`.

26.5.4 String construction on Windows systems

When LispWorks for Windows starts up on a OS with a non-Latin-1 code page, it calls

```
(set-default-character-element-type 'cl:character)
```

so that by default, newly constructed strings can contain the data likely to be returned from the OS or user input.

If you know your string only needs to contain 8-bit data, then you can create it explicitly with element type `base-char`.

Conversely if you know that a string may need to contain 16-bit data even on a Latin-1 code page system, then you should create it explicitly with element type `bmp-char` (or `cl:character` if 32-bit data is needed).

26.6 External Formats

External formats are two-way translations from Lisp’s internal encoding to an external encoding. They can be used in file I/O, and in passing and receiving string data in foreign function calls.

An external format is named in LispWorks by an external format specification (ef-spec). An ef-spec is a symbol naming the external format, or a list with such a name as its first element followed by parameter/value pairs.

LispWorks has a number of predefined external formats:

- `win32:code-page`
  The Windows code page with identifier given by the :id parameter. Implemented only on Windows.

- `:latin-1`
  ISO8859-1.

- `:latin-1-terminal`
  As Latin-1, except that if a non-Latin-1 character is output, it is written as `<xxxx>` where `xxxx` is the hexadecimal character code and does not signal error.
### :latin-1-safe

As Latin-1, except that if a non-Latin-1 character is output, it is written as ? and does not signal error.

### :macos-roman

The Mac OS Roman encoding.

### :ascii

ASCII.

### :unicode :utf-16

:utf-16 with default native byte order. See “16-bit External formats guide” on page 445 for details and variants.

**Compatibility note:** In LispWorks 6.1 and earlier versions, :unicode encodes 16-bit characters reading.

### :utf-8

The UTF-8 encoding of Unicode.

### :utf-16

The UTF-16 encoding of Unicode with big-endian byte order. See “16-bit External formats guide” on page 445 for details and variants.

### :utf-32

The UTF-32 encoding of Unicode with big-endian byte order.

**Note:** There is a :utf-32 external format corresponding to each of the :utf-16 variants.

### :bmp

Reads and writes 16-bit characters with native byte order. See “16-bit External formats guide” on page 445 for details and variants.

### :jis

JIS. The encoding data is read from a file Uni2JIS and is pre-built into LispWorks.

**Note:** Uni2JIS is provided by way of documentation in the directory lib/7-1-0-0/etc/. It is also used at run time by the function cl:char-name.

### :euc-jp

EUC-JP. The encoding data is read from a file Uni2JIS and is pre-built into LispWorks.

### :sjis

Shift JIS.

### :windows-cp936
26.7 16-bit External formats guide

26.7.1 Unicode

In LispWorks 6.1 and earlier versions the external format :unicode is actually "raw UCS-2", that is reading and writing only 16-bit characters (including character objects corresponding to surrogate code points). The :unicode format now maps to :utf-16 with the native endianness (by default). This interprets surrogate code points (#xd800 to #xdf7f) differently: the old :unicode would read these as if they are actual characters, while :utf-16 and hence the new :unicode will try to interpret them as encoding supplementary characters (codes #x10000 to #x10ffff). The latter behavior is probably what you need, so in most cases there is no need to replace usage of :unicode. There is no external format which interprets surrogate code points as characters now, but you can uses any of the :bmp formats with :use-replacement t to read 16-bit characters without giving an error, although this does not exactly match the input, because surrogate code points are translated by the replacement character. The only format that can read anything without any loss is :latin-1.

Note that :unicode differs from :utf-16 by the default byte order that it uses: :utf-16 defaults to big-endian (matching the Unicode standard), while :unicode defaults to the native byte order.
26.7.2 UTF-16

There are now several UTF-16 external formats. There are more than one because UTF-16 is actually two different encodings: UTF-16 big-endian and UTF-16 little-endian.

:utf-16-native and :utf-16-reversed are the actual implementation formats. They implement UTF-16 with the native byte order (:utf-16-native) or the reversed byte order (:utf-16-reversed).

:utf-16be and :utf-16le implement the big-endian (:utf-16be) and little-endian (:utf-16le) UTF-16. The system maps these formats to :utf-16-native or :utf-16-reversed as appropriate, depending on the byte order of the computer.

:utf-16 implements the UTF-16 standard, defaulting to UTF-16BE unless there is a BOM (Byte Order Mark).

In general, you will need to decide which of these to use depending on the circumstances.

26.7.3 BMP

There are now a few BMP external formats, which read and write only 16-bit characters (characters in the range 0 to #xffff, excluding the surrogate range #xd800 to #xdfff).

:bmp-native and :bmp-reversed are the actual implementation formats. They implement reading 16-bit characters with the native byte order (:bmp-native) or the reversed byte order (:bmp-reversed). These formats never read supplementary characters. When they encounter a surrogate code point, they either signal an error or replace it by the replacement character, depending on the parameter :use-replacement.

:bmp implements 16-bit character reading and writing, defaulting to the native one.

Notes: In LispWorks 6.1 and earlier versions, the :unicode external format is similar to :bmp now, but handles surrogate code points as if they represent
characters. In LispWorks 7.0 and later, :unicode maps to :utf-16, and there is no external format that reads surrogate code points as characters.

26.8 External Formats and File Streams

The :external-format argument of open and related functions should be an ef-spec, where the name can be :default. The symbol :default is the default value.

If you know the format of the data when doing file I/O, you should definitely specify external-format explicitly, in the ef-spec syntax described in this section.

26.8.1 Complete external format ef-specs

An ef-spec is "complete" if and only if the name is not :default and the parameters include :eol-style.

All external formats have an :eol-style parameter. If eol-style is not explicit in an ef-spec a default is used. The allowed values are

:lf This is the default on non-Windows systems, meaning that lines are terminated by Linefeed.
:crlf This is the default on Windows, meaning that lines are terminated by Carriage-Return followed by Linefeed.
:cr Lines are terminated by Carriage-Return.

26.8.2 Using complete external formats

If open or with-open-file gets a complete :external-format argument then, it is used as is. For example, this form opens an ASCII linefeed-terminated stream:

(with-open-file (ss "C:/temp/ascii-lf" :direction :output :external-format '(:ascii :eol-style :lf))
(stream-external-format ss))
=> (:ASCII :EOL-STYLE :LF)
If you know the encoding of a file you are opening, then you should pass the appropriate :external-format argument.

### 26.8.3 Guessing the external format

If `open` or `with-open-file` gets a non-complete :external-format argument `ef-spec` then the system decides which external format to use by calling the function `guess-external-format`.

The default behavior of `guess-external-format` is as follows:

1. When `ef-spec`’s name is :default, this finds a match based on the filename; or (if that fails), looks in the Emacs-style (-*--) attribute line for an option called ENCODING or EXTERNAL-FORMAT or CODING; or (if that fails), chooses from amongst likely encodings by analysing the bytes near the start of the file, or (if that fails) uses a default encoding. Otherwise `ef-spec`’s name is assumed to name an encoding and this encoding is used.

2. When `ef-spec` does not include the :eol-style parameter, it then also analyzes the start of the file for byte patterns indicating the end-of-line style, and uses a default end-of-line style if no such pattern is found.

The file in this example was written by a Windows program which writes the Byte Order Mark at the start of the file, indicating that it is Unicode encoded. The routine in step 1 above detects this:

```lisp
(set-default-character-element-type 'character)
=> CHARACTER
(with-open-file (ss "C:/temp/unicode-notepad.txt")
  (stream-external-format ss))
=> (:UNICODE :LITTLE-ENDIAN T :EOL-STYLE :CRLF)
```

The behavior of `guess-external-format` is configurable via the variables *file-encoding-detection-algorithm* and *file-eol-style-detection-algorithm*. See the manual pages for details.
26.8.3.1 Example of using UTF-8 by default

To change the default for all file access via open, compile-file and so on, you can modify the value of *file-encoding-detection-algorithm*.

For example given the following definition:

```lisp
(defun utf-8-file-encoding (pathname ef-spec buffer length)
  (declare (ignore pathname buffer length))
  (system:merge-ef-specs ef-spec :utf-8))
```

then this makes it use UTF-8 as a fallback:

```lisp
(setq system:*file-encoding-detection-algorithm*
      (substitute 'utf-8-file-encoding
                  'system:locale-file-encoding
                  system:*file-encoding-detection-algorithm*))
```

and this forces it to always use UTF-8:

```lisp
(setq system:*file-encoding-detection-algorithm*
      '(utf-8-file-encoding))
```

26.8.3.2 Example of using UTF-8 if possible

The example in “Example of using UTF-8 by default” on page 449 will use UTF-8 even if the file is known to contain bytes that cannot be in this encoding. As an alternative way to use UTF-8 when possible, you can modify the value of *specific-valid-file-encodings*.

For example:

```lisp
(pushnew :utf-8 system:*specific-valid-file-encodings*)
```

26.8.4 External formats and stream-element-type

The :element-type argument in open and with-open-file defaults to the value of *default-character-element-type*.

If element-type is not :default, checks are made to ensure that the resulting stream’s stream-element-type is compatible with its external format:

1. If direction is :input or :io, the element-type argument must be a supertype of the type of characters produced by the external format.
2. If \textit{direction} is \texttt{:output} or \texttt{:io}, the \textit{element-type} argument must be a sub-type of the type of characters accepted by the external format.

If the \textit{element-type} argument does not satisfy these requirements, an error is signaled.

If \textit{element-type} is \texttt{:default} the system chooses the \texttt{stream-element-type} on the basis of the external format.

26.8.5 External formats and the LispWorks Editor

The LispWorks Editor uses \texttt{open} with \texttt{:element-type :default} to read and write files. On reading a file, the external format is remembered and used when saving the file. On writing a Unicode (UTF-16) file, the Byte Order Mark is written.

It is possible to insert characters in the Editor (for example by pasting clipboard text) which are not supported by the chosen external format. This will lead to errors on attempt to save the buffer. You can handle this by setting the external format appropriately.

See the \textit{LispWorks Editor User Guide} for more details.

26.8.6 Byte Order Mark

The Unicode Byte Order Mark (BOM) is treated as whitespace in the default readtable. This allows the Lisp reader to read a 16-bit (UTF-16 or BMP encoded) file regardless of whether the BOM is present. See “16-bit External formats guide” on page 445 for more information.

Some editors including Microsoft Notepad and the LispWorks editor write the BOM when writing a file with 16-bit (UTF-16 or BMP) encoding.

26.9 External Formats and the Foreign Language Interface

External formats can be used to pass and receive string data via the FLI. See the section on string types in the \textit{LispWorks Foreign Language Interface User Guide and Reference Manual}. 
26.10 Unicode character and string functions

This section lists functions which compare characters and strings similarly to \texttt{cl:char-equal}, \texttt{cl:string-greaterp} and so on, but which use Unicode's simple case folding rules.

There are also predicates for properties of characters in Unicode's "general category", corresponding to \texttt{cl:alpha-char-p}, \texttt{cl:both-case-p} and so on.

26.10.1 Unicode case insensitive character comparison

The functions \texttt{unicode-char-equal}, \texttt{unicode-char-not-equal}, \texttt{unicode-char-lessp}, \texttt{unicode-char-not-lessp}, \texttt{unicode-char-greaterp} and \texttt{unicode-char-not-greaterp} compare characters similarly to \texttt{cl:char-equal} etc, but using Unicode's simple case folding rules.

26.10.2 Unicode case insensitive string comparison

The functions \texttt{unicode-string-equal}, \texttt{unicode-string-not-equal}, \texttt{unicode-string-lessp}, \texttt{unicode-string-not-lessp}, \texttt{unicode-string-greaterp} and \texttt{unicode-string-not-greaterp} compare strings similarly to \texttt{cl:string-equal} etc, but using Unicode's simple case folding rules.

26.10.3 Unicode character predicates

The predicates \texttt{unicode-alphanumeric-p}, \texttt{unicode-alpha-char-p}, \texttt{unicode-lower-case-p}, \texttt{unicode-upper-case-p} and \texttt{unicode-both-case-p} test for properties of a character in Unicode's "general category".
This chapter describes the interfaces which provide information about the environment in which LispWorks is running. This includes the operating system, the file system, the physical location of the LispWorks executable, and the arguments it was passed on startup.

### 27.1 The Operating System

The Common Lisp function `software-type` returns a generic name for the Operating System. The Common Lisp function `software-version` returns information about the version of the Operating System.

In particular, `software-type` cannot be used to distinguish between different versions of Windows, whereas `software-version` allows you to identify variants such as Windows Vista, Windows 7, Windows 8, Windows 10 and so on. See the manual pages for details.

### 27.2 Site Name

The Common Lisp functions `short-site-name` and `long-site-name` can be configured using `setf`:

```
(setf (long-site-name) "LispWorks Ltd"
      (short-site-name) "LW")
```
27.3 The Lisp Image

The function `lisp-image-name` returns the namestring of the full path of the LispWorks executable or dynamic library (DLL). For example, the directory of the image can be found using:

```
(pathname-location (lisp-image-name))
```

To create a new executable or DLL, typically after loading patches, modules and application code, use `save-image` or `deliver`.

**Note:** Microsoft Windows supports Long and Short forms of paths. You may need to convert a namestring using `long-namestring` or `short-namestring`.

27.4 The Command Line

The command line used to run LispWorks can be found using the variable `*line-arguments-list*`. The value is a list of strings containing the executable name followed by any other command line arguments, in the order they were passed.

For example, if your application needs to behave differently when passed an argument `-foo`, use the following test:

```
(member "-foo" sys:*line-arguments-list* :test 'string)
```

27.4.1 Command Line Arguments

The following command line options are supported by the system.

`-build build-script`

Typically this is used for the purpose of building another image.

`build-script` can name a file to be loaded on startup. This file will be the build script which loads your code and calls `save-image` or `deliver`. LispWorks quits after loading the file. If an error is signaled while loading the file, a backtrace is displayed and LispWorks quits.
build-script can also be -, a single minus sign. Passing -build - causes LispWorks to read and execute a build script from stdin. This is useful if you want to embed a build script within a shell script that runs LispWorks, for example:

```
(lispworks-7-1-0-x86-linux -build - <<END
(write-line "This is the build script.")
END)
```

Note that this technique using <<END does not work on Microsoft Windows.

An image run with -build runs itself, and not the default saved session if you created one. See “Saved sessions” on page 182 for information on saved sessions.

- build calls load-all-patches automatically. There is no harm if your build script also calls load-all-patches.

- environment

Start the LispWorks IDE development environment automatically, even in an image saved with (save-image ... :environment nil)

- eval form

Evaluates the Lisp form form before loading initialization files.

If form requires multiprocessing, then change it to push a process specification onto *initial-processes*. This will delay evaluation until multiprocessing has started, either by the -multiprocessing command line options or because LispWorks was saved to start multiprocessing.

- env

A synonym for -environment.

- display display

Sets the X display to use when starting a LispWorks GUI on X Windows.

- IIOPHost host
Controls the host name in placed in IORs. See *Developing Component Software with CORBA* for details.

-IIOIPnumeric

IORs contain a host name which is the numeric IP address obtained by reverse lookup of the machine name. See *Developing Component Software with CORBA* for details.

-init init-file

*init-file* names a file to be loaded on startup after *siteinit-file*. The file is user’s own LispWorks initialization file, containing code that by default is loaded when LispWorks is started. It is useful for loading initializations that should not be done for all users.

Initially the default is to load the file "~/.lispworks" where ~ expands to the user’s home directory as described in “Configuration and initialization files” on page 178.

Your default initialization file can be set in the LispWorks IDE. See “Setting Preferences” in the *LispWorks IDE User Guide* for details.

If *init-file* is not found, an error is signaled. To suppress loading of a user initialization file, pass -init -.

-load file

Loads the file *file* before loading initialization files.

-lw-no-redirection

Makes the supplied image run itself, and not the default saved session if you created one. See “Saved sessions” on page 182 for information on saved sessions.

-multiprocessing

Initializes multiprocessing on startup. See Chapter 19, “Multiprocessing”.

-no-restart-function
Suppresses the execution of a restart function on startup. Restart functions can be supplied when saving an image to automatically invoke application code. This argument suppresses that behavior. See `save-image`, page 1001.

```
-ORBport orbport
```

`orbport` specifies a port number for the LispWorks ORB. The special value 0 allows the system to pick a port.

```
--relocate-image BaseAddress
```

Causes the image to relocate at `BaseAddress` on supported platforms, as described in “Startup relocation” on page 459. This can be useful on a system where libraries are mapped in address space that LispWorks would otherwise use as it grows. If the image is saved, then on restart without `--relocate-image`, it will locate itself automatically at `BaseAddress`.

**Compatibility note:** In LispWorks 5.0 and earlier versions, to be effective, `--relocate-image` must be the first argument on the LispWorks command line. This restriction does not apply in LispWorks 7.1.

```
--reserve-size ReserveSize
```

Specifies the reserve size on supported platforms, as described in “Startup relocation” on page 459.

```
-siteinit siteinit-file
```

`sitestrinit-file` names a file to be loaded on startup. The file is the LispWorks site initialization file, containing code that by default is loaded when LispWorks is started by any user in that installation. The default is to load the file that is the result of evaluating

```
(sys:lispworks-file "config/siteinit.lisp")
```

If `siteinit-file` is not found, an error is signaled. To suppress loading of a site initialization file, pass `-siteinit -`.
27.4.2 Accessing environment variables

Use `environment-variable` get and set the value of an environment variable in the environment table of the OS process that called LispWorks.

To remove FOO from the environment table do:

```lisp
(setf (lw:environment-variable "FOO") nil)
```

27.5 Address Space and Image Size

There are two factors that affect the maximum size of the Lisp image: the size of real memory, and the layout of memory. On most platforms you can relocate LispWorks to avoid clashes with other software as described in “Startup relocation” on page 459.

27.5.1 Size of real memory

If LispWorks becomes significantly larger than the size of the real memory, then paging will be the main activity and LispWorks will not function effectively.

27.5.2 Layout of memory

This is Operating System-dependent:

On Solaris, 32-bit LispWorks is mapped at \texttt{#x10000000}. In principle it can grow to almost \texttt{#x80000000} (the libraries are at higher addresses).

For the other platforms and for 64-bit LispWorks, see the discussion in “Startup relocation” on page 459.

27.5.3 Reporting current allocation

The simplest way to see the current Lisp allocation is to call \texttt{(room t)}.

To obtain values representing the current total allocation, call \texttt{room-values}. 

27.6 Startup relocation

On startup, LispWorks normally maps its heap at the address where it was mapped when the image was saved. It maps more memory close to this when needed. This may cause memory clashes with other software, but such clashes may be avoided by relocating LispWorks.

32-bit LispWorks is relocatable on Microsoft Windows, Intel Macintosh, Linux, x86/x64 Solaris, FreeBSD and AIX. 64-bit LispWorks is relocatable on all supported platforms. The discussion in this section is applicable to all relocatable implementations.

On Microsoft Windows and Macintosh, LispWorks detects memory clashes and avoids them automatically. On these platforms there is no need to explicitly relocate LispWorks. The other relocatable implementations - LispWorks (32-bit) for Linux, LispWorks (64-bit) for Linux, LispWorks (32-bit) for FreeBSD, LispWorks (64-bit) for FreeBSD, LispWorks (32-bit) for x86/x64 Solaris, LispWorks (64-bit) for x86/x64 Solaris, LispWorks (64-bit) for SPARC/Solaris, LispWorks (32-bit) for AIX and LispWorks (64-bit) for AIX - cannot safely detect memory clashes. Relocation may therefore be useful in these implementations.

27.6.1 How to relocate LispWorks

Relocate LispWorks by passing two parameters: the base address and the reserve amount. Both are optional. The interpretation of these parameters is very different between 64-bit LispWorks and 32-bit LispWorks.

To relocate a LispWorks executable, pass one or both of these command line arguments:

```bash
--relocate-image BaseAddress
```

The base address, interpreted as a hexadecimal number by calling `strtol(BaseAddress, NULL, 16)`

```bash
--reserve-size ReserveSize
```

The reserve size, interpreted as a hexadecimal number by calling `strtol(ReserveSize, NULL, 16)`
On all relocatable platforms, a LispWorks dynamic library or Windows DLL can be relocated by calling `InitLispWorks` with second and/or third argument non-zero.

On non-Windows platforms, you can add the appropriate call to `InitLispWorks` in wrappers written in C and added to the dynamic library by passing `dll-added-files` to `save-image` or `deliver`. There is no such option in LispWorks for Windows.

The startup relocation takes some time, normally less than 0.1 seconds on a modern machine. If the relocation address is fixed and known, this startup overhead can be eliminated by relocating the image before calling `save-image` or `deliver`.

### 27.6.2 Startup relocation of 32-bit LispWorks

32-bit LispWorks on x86 platforms maps its heap in one continuous block, and then grows upwards from the top. When it reaches a region that it cannot use, it can skip it. On Windows and Macintosh this skipping is safe, because LispWorks can safely detect regions of memory that it cannot use. On other x86 platforms, both the initial mapping and the further growth cannot safely detect when they overwrite some other code.

`BaseAddress` (passed on command line with `--relocate-image` or as the second argument to `InitLispWorks`) tells LispWorks where to map the heap. On Windows and Macintosh, if the address is already used the heap will be mapped elsewhere. On other platforms, the mapping always works, and may destroy what is already mapped at that address.

`ReserveSize` (passed on command line with `--reserve-size` or as the third argument to `InitLispWorks`) tells LispWorks how much additional memory to reserve. Reservation is properly supported on Windows and Macintosh, though the actual reserved size can be smaller if it fails to reserve as much as was requested. On platforms that do not support reservation (that is, not Windows or Macintosh), the reservation is done by using `mmap` with protection `PROT_NONE`.

For a description of the memory layout on each platform, see “Memory layout” on page 140.
27.6.3 Startup relocation of 64-bit LispWorks

The size of address space that 64-bit LispWorks can use is limited by the size of internal tables to a "span" of $2^{44}$ (16TB). The span always starts at 0.

Inside this span LispWorks can use any address. However, to avoid clashes with other software, it uses memory only in some defined range.

Startup relocation means changing this range. *BaseAddress* (passed on command line with `--relocate-image` or as the second argument to *InitLispWorks*, rounded up to $2^{28}$) is the start of the range. *ReserveSize* (passed on command line with `--reserve-size` or as the third argument to *InitLispWorks*) is the size of the range. The default of the size of the range is $2^{40}$.

If the entire heap is within the new range, nothing else is done. If some part of the heap is outside the new range, the heap is relocated.

The range in each 64-bit LispWorks implementation starts at $\#x4000000000$ (256 GB).

27.6.3.1 Linux

On old Linux systems LispWorks (64-bit) for Linux has range 192 GB, ending at $\#x7000000000$, because old Linux systems cannot map above $\#x8000000000$ and put the dynamic libraries just below that limit (at least in some configurations). Since LispWorks uses the address space sparsely, it will run out of memory with less virtual memory, probably around 150 GB to 160 GB. If more memory is required, the range can be extended downwards, and possibly some distance upwards too. If other software uses memory in the range from $\#x4000000000$ to $\#x7000000000$, LispWorks should be relocated (potentially just by decreasing the range) to avoid memory clashes.

Modern Linux systems have a much larger address space and the default size of the LispWorks range is $\#x4000000000$ (4TB).

27.6.3.2 SPARC Solaris

In LispWorks (64-bit) for SPARC Solaris the default range is 768 GB, ending at $\#x1000000000$. If other software uses memory in this range, the range for LispWorks should be decreased to avoid memory clashes.
27.6.3.3 Windows, Macintosh and AIX

In LispWorks (64-bit) for Windows, LispWorks (64-bit) for Macintosh and LispWorks (64-bit) for AIX the size of the range is #x3c00000000 (3.75TB). Since these platforms properly support reservation, there should not be any reason to change the range. The only time when this is needed is when other software insists on using some address in this range and does not relocate automatically.

27.7 Calling external programs

You can call an external program using call-system, call-system-showing-output and open-pipe.

You can call C programs using the FLI. See the LispWorks Foreign Language Interface User Guide and Reference Manual.

On Microsoft Windows a COM/Automation interface is provided. See the LispWorks COM/Automation User Guide and Reference Manual. There is also a DDE interface - see Chapter 22, “Dynamic Data Exchange”.


27.7.1 Interpreting the exit status

call-system returns the exit status of the process it created, and potentially a signal number. Similarly pipe-exit-status can query the exit status and signal number associated with a process that was created by open-pipe.

On Unix-like systems when using a string as the command with a typical shell, the exit status is the exit status of the command that is executed. If it is an actual executable (rather than a built-in command) it is the exit status of the process that invoked by this executable. That is not always reliable. In a typical shell you can precede the last command by the word exec to cause the shell to replace itself by the executable, and then the return value is guaranteed to be from the executable. On Microsoft Windows and when not using string as a command, there is only one process and the exit status is the exit status of this process.
On Unix-like systems, on normal exit the exit status is the argument that was passed to the C function `exit` (or `_exit`) or the value returned from the main function, and the signal number is `nil`. To interpret the normal exit status you need to know what the process does. Normally 0 means success. If the process exited as a result of a signal then the second return value gives the number of the signal.

On Windows, the exit status is either the argument to `ExitProcess` or `TerminateProcess`, the return value of `main` or `WinMain`, or an exception value.

### 27.8 Snapshot debugging of startup errors

When an error occurs during initialization (for example, because of code in an initialization file) and the image is configured to start the LispWorks IDE, by default it catches the error, starts the IDE and displays the error in a snapshot debugger.

You should note that because this is a snapshot, you cannot actually continue or abort or return from a frame. The snapshot debugger is simply a tool to help debugging the error.

The behavior is controlled by the variable `*debug-initialization-errors-in-snap-shot*`.

### 27.9 System message log

The system message log is used by the system to produce messages that indicate that something is not as expected, where this is not an error. You can manipulate the log with `set-system-message-log`.

### 27.10 Exit status

You can return a process exit status to the Operating System when LispWorks or a delivered LispWorks application quits.

Do this by passing a `status` value to the function `quit`. For example:

```
(quit :status 42)
```
27.11 Creating a new executable with code preloaded

There are two ways to create a new executable with your code preloaded.

- To write a copy of the currently running image to disk, use `save-image`, page 1001. The saved image requires a development license key to run.
- To create a runtime image, removing unused code to make the image smaller, call `deliver`. For more details see the *LispWorks Delivery User Guide*.

For example of how to use `save-image`, see the section “Saving and testing the configured image” in the *LispWorks Release Notes and Installation Guide*.

See “Code signing in saved images” on page 181 for information about code signing your new executable.

27.12 User Preferences

LispWorks provides an API for setting and querying persistent per-user settings in a platform-dependent registry.

27.12.1 Location of persistent settings

On Microsoft Windows the preferences are stored in the HKEY_CURRENT_USER branch of the Windows registry. (LispWorks also offers a general Windows registry API, described in “Accessing the Windows registry” on page 468.)

On non-Windows the preferences are stored in subdirectories of the user’s home directory.

To implement preferences for your LispWorks application, you will need to define a registry path using `(setf product-registry-path)` and read it using `product-registry-path`.

27.12.2 Accessing persistent settings

Get and set preferences under the product path at run time with `user-preference` and `(setf user-preference)`.
27.12.3 Example using user preferences

Define a registry path:

```lisp
(setf (sys:product-registry-path :deep-thought)
  '("Software" "My Company" "Deep Thought"))
```

Store a preference for the current user:

```lisp
(setf (user-preference "Answers"
  "Ultimate Question"
  :product :deep-thought) 42)
```

Retrieve a preference for the current user, potentially in a subsequent session:

```lisp
(user-preference "Answers" "Ultimate Question"
  :product :deep-thought)
```

27.13 File system interface

27.13.1 Fast access to files in a directory

`fast-directory-files` gives a faster way to access files than `directory`, especially in situations when you need to filter based on simple features such as size and access time, or filter based on the name in a more complex way than `directory` can.

Instead of creating a list of pathnames and returning it, `fast-directory-files` traverses the files and calls a callback function on each file with its name and an opaque handle, which is referred to as `fdf-handle`. From this handle, you can retrieve the size, last-access time and last-modify time, and query whether the file is a directory, whether it is a link (for platforms other than Windows), and whether it is writable. The implementation makes the access to the `fdf-handle` much faster than doing the same by calling `directory` and then calling `cl:file-write-date` and similar functions on the result.

When the callback returns non-nil, `fast-directory-files` collects the filename, otherwise it ignores it. Hence the callback can be used both as a filter and to actually do some work. In many cases, the callback will always return `nil`, and the call will be used just to map the callback on the file for the "side-effects" of the callback.
fast-directory-files is restricted to one directory level, that is it cannot deal with wild directories.

27.14 Special locations in the file system
This section describes interfaces allowing you to identify and access various special locations in the file system.

27.14.1 The home directory
This section describes the implementation of the Common Lisp function cl:user-homedir-pathname.

On Unix-based systems, the home directory is looked up using the C function getpwuid.

On Microsoft Windows systems, cl:user-homedir-pathname uses the environment to construct its result. It uses the values of the environment variables HOMEDRIVE and HOMEPATH, if both are defined. If at least one of environment variables HOMEDRIVE and HOMEPATH is not defined, then a pathname #P"C:/users/login-name" is returned. These environment variables should be correctly set before LispWorks starts. However it is possible to change the values in Lisp using

(setf environment-variable)


27.14.2 Special Folders
On Microsoft Windows, Mac OS X and Android there are various special folders used for application data and user data. Here are some examples of the folder for application data which is shared between all users.

Windows 10, Windows 8, Windows 7 and Windows Vista
C:\ProgramData

Windows XP (now unsupported)
27.14 Special locations in the file system

The result of calling `getExternalFilesDir` on the application context with `null`.

The locations and folder names can differ between versions of the operating system, therefore it is useful to have a system-independent way to get the path at run time. The function `get-folder-path` can be used to retrieve the path to special folders. Directory pathnames corresponding to each of the examples above can be obtained by calling:

```lisp
(sys:get-folder-path :common-appdata)
```

Here is another example of differences between operating systems. On Windows 7 and Windows Vista:

```lisp
(sys:get-folder-path :my-documents)
=> #P"C:/Users/dubya/Documents/"
```

On Mac OS X:

```lisp
(sys:get-folder-path :my-documents)
=> #P"/u/ldisk/dubya/Documents/"
```

See `get-folder-path` for more details.

On Microsoft Windows there is a profile folder for each user. You can find the profile path for the current user with the function `get-user-profile-directory`.

27.14.3 Temp files

A "temp file" is a file in the "temp directory" which is guaranteed to be new. Its name contains a random element.

Create a temp file by calling either of the functions `open-temp-file` and `create-temp-file`. For example:
On Microsoft Windows:

```
(create-temp-file :prefix "LW")
=> #P"C:/DOCUME-1/dubya/LOCALS-1/Temp/LW383vWfZC.tmp"
```

On Linux:

```
(create-temp-file :prefix "LW")
=> #P"/tmp/LWadokNa.tmp"
```

The function `set-temp-directory` allows you to set the "temp directory", that is the default directory used for temp files.

### 27.15 Accessing the Windows registry

There is an API for accessing the registry on Microsoft Windows. It is available only in LispWorks for Windows. All of its symbols are in the `win32` package.

Create and delete keys with the functions `create-registry-key` and `delete-registry-key`. Open a key for reading and/or writing with `open-registry-key` and close it with `close-registry-key`, or wrap your registry operation inside the macro `with-registry-key`.

Query the registry with `registry-key-exists-p`, `enum-registry-value`, `collect-registry-values`, `collect-registry-subkeys`, `query-registry-key-info`, `query-registry-value`, and `registry-value`. Write to the registry with `set-registry-value` or `(setf registry-value)`.

For example, this function returns the name, progid and filename for each of the installed ActiveX controls:
(defun collect-control-names (&key insertable
  (max-name-size 256)
  (max-names most-positive-fixnum))
  (win32:collect-registry-subkeys
   "CLSID"
   :root :root
   :max-name-size max-name-size
   :max-names max-names
   :value-function
   #'(lambda (hKeyClsid ClassidName)
       (win32:with-registry-key
         (hkeyX ClassidName :root hKeyClsid :errorp nil)
         (when (and
             (win32:registry-key-exists-p "Control"
               :root hkeyX)
             (if insertable
                 (win32:registry-key-exists-p "Insertable"
                   :root hkeyX)
                 t))
         (when-let
           (progid (win32:query-registry-value "ProgID" nil
                   :root hkeyX
                   :errorp nil))
           (values
            (list
             (win32:query-registry-value nil nil
               :root hkeyX)
             progid
             (win32:query-registry-value "InprocServer32" nil
               :root hkeyX
               :errorp nil))
            t)))))))

27.16 Pathname comparison on Mac OS X

Comparing pathnames using `equal` and `equalp` is case-sensitive on the Macintosh. This can lead to occasional unexpected mismatch of pathnames, because the HFS+ filesystem is usually case-insensitive (some Macintosh file systems are case-sensitive).
This chapter describes miscellaneous functionality which does not belong in other chapters.

### 28.1 Object addresses and memory

In general, you cannot rely on the addresses of Lisp objects, because the Garbage Collector moves objects. The functions described in this section are intended for debugging only.

You can find the current address of a Lisp object as an integer by `object-address`. You can get the pointer of an object (current address plus any tagging) by `object-pointer`. This is what is normally used when printing objects unreadably. You can find which object is currently at some address by using `pointer-from-address`.

You can find the size of a heap object by using `find-object-size`. However, many Lisp objects are made of multiple heap objects, and typically the "root" heap object (the one that the Lisp pointer points to) is relatively small, so for these objects `find-object-size` returns a meaningless value. It is actually useful only for vectors that are simple (not with fill pointer or adjustable or displaced) and structures. It also gives meaningful values for integers, floats and conses.
28.2 Optimized integer arithmetic and integer vector access

This section describes ways to perform certain operations as efficiently as possible, including vector access and raw 32-bit arithmetic. Additionally in 64-bit LispWorks, raw 64-bit arithmetic is possible.

28.2.1 Typed aref vectors

You can make vectors of certain element types which allow the most efficient access possible when compiled with suitable optimize qualities.

To do this:

1. Make a vector with `make-typed-aref-vector`.

2. Access the vector using `typed-aref` and `(setf typed-aref)` with a type argument of `double-float`, `float`, `single-float`, `int32`, `(unsigned-byte n)` or `(signed-byte n)` where \(n = 8, 16\) or \(32\).

   Additionally, in 64-bit LispWorks the types `(unsigned-byte 64)` and `(signed-byte 64)` are supported.

3. Compile the access with `safety 0` (and for float types, `float 0`) and a constant `type`.

See `typed-aref` for more details and examples.

Efficient access to foreign arrays is also available. See `fli:foreign-typed-aref` in the *LispWorks Foreign Language Interface User Guide and Reference Manual*.

28.2.2 Fast 32-bit arithmetic

The INT32 API provides a way to perform optimal raw 32-bit arithmetic. Note that, unlike Lisp integer types, this is modulo \(2^{32}\) like the C `int` type.

The INT32 symbols are all in the `system` package.

The Lisp type `int32` reads 32 bits of memory, like `(signed-byte 32)`, but the data is in `int32` format for use with the INT32 API.
28.2 Optimized integer arithmetic and integer vector access

28.2.2.1 Optimized and unoptimized INT32 code

When optimized correctly, the intermediate int32 objects are not constructed. In unoptimized code, sequences of operations like

\[(\text{sys:int32}+ (\text{sys:int32}- a \ b) (\text{sys:int32}- c \ d))\]

will generate intermediate int32 objects for the results of the subtraction, but the compiler can optimize these away because it knows that the function int32+ consumes int32 objects.

Note: the INT32 API is not designed to optimize sys:int32 objects passed as arguments.

28.2.2.2 The INT32 API

The INT32 API contains the type int32, a vector type simple-int32-vector and accessor, functions to convert int32 to and from integer, some constant int32 values, and a full range of operators for mod $2^{32}$ arithmetic.

You can find all these by evaluating

\[(\text{apropos "INT32" "SYSTEM" t})\]

For details for each, see the entries starting with int32 in Chapter 49, “The SYSTEM Package”.

28.2.2.3 INT32 Optimization

The optimization works safely but without boxing when possible. You need

\[(\text{optimize (float 0))}\]

to get the optimization. This float level affects whether INT32 operations are optimized. This declaration must be placed at the start of a function (not on an inner let or locally form).

In this example the safety level assures a second optimization in fli:foreign-typed-aref:
(defun incf-signed-byte-32 (ptr index)
  (declare (optimize (safety 0) (float 0))
    (type fixnum index))
  (setf (fli:foreign-typed-aref 'sys:int32 ptr index)
        (sys:int32-1+ (fli:foreign-typed-aref 'sys:int32
                      ptr index)))

;; return ptr, since otherwise the int32 would
;; need to be boxed to return it
ptr)

28.2.3 Fast 64-bit arithmetic

The INT64 API provides a way to perform optimal raw 64-bit arithmetic. Note
that, unlike Lisp integer types, this is modulo 2^64 like the C long long or
int64 types.

The INT64 symbols are all in the system package.

The Lisp type int64 reads 64 bits of memory, like (signed-byte 64), but the
data is in int64 format for use with the INT64 API.

28.2.3.1 Optimized and unoptimized INT64 code

When optimized correctly, the intermediate int64 objects are not constructed.
In unoptimized code, sequences of operations like

(sys:int64+ (sys:int64- a b) (sys:int64- c d))

will generate intermediate int64 objects for the results of the subtraction, but
the compiler can optimize these away because it knows that the function
int64+ consumes int64 objects.

Note: the INT64 API is not designed to optimize sys:int64 objects passed as
arguments.

28.2.3.2 The INT64 API

The INT64 API contains the type int64, a vector type simple-int64-vector
and accessor, functions to convert int64 to and from integer, some constant
int64 values, and a full range of operators for mod 2^64 arithmetic.

You can find all these by evaluating
28.2 Optimized integer arithmetic and integer vector access

(apropos "INT64" "SYSTEM" t)

For details for each, see the entries starting with int64 in Chapter 49, “The SYSTEM Package”.

28.2.3.3 INT64 Optimization

INT64 optimization occurs only in 64-bit LispWorks. The INT64 API is not optimized in 32-bit LispWorks.

The optimization works safely but without boxing when possible. You need

(optimize (float 0))

to get the optimization. This float level affects whether INT64 operations are optimized. This declaration must be placed at the start of a function (not on an inner let or locally form).

In this example the safety level assures a second optimization in fli:foreign-typed-aref:

(defun incf-signed-byte-64 (ptr index)
  (declare (optimize (safety 0) (float 0))
           (type fixnum index))
  (setf (fli:foreign-typed-aref 'sys:int64 ptr index)
        (sys:int64-1+ (fli:foreign-typed-aref 'sys:int64 ptr index)))
  ;; return ptr, since otherwise the int64 would
  ;; need to be boxed to return it
  ptr)

28.2.4 Integer vector access

octet-ref and base-char-ref (and their setters) are provided to allow efficient access to simple vectors of element type (unsigned-byte 8) or base-char (that is, simple-base-strings) in the same code.

Other vector types are accepted, but for these specific string and binary vector types octet-ref and base-char-ref match what aref and (setf aref) do except that they always take and return the same value/result type, and they are also more efficient than aref.

Use octet-ref and base-char-ref according to whether you work with elements of type integer or base-char.
28.3 Transferring large amounts of data

You can write Lisp data in a binary format to a file using `dump-forms-to-file` or `with-output-to-fasl-file` with `dump-form`. The file can then be loaded by `load-data-file`.

This allows you to transfer large amounts of data without using the Lisp printer and reader, which is much more efficient and robust.

28.4 Rings

Ring objects can be used to hold Lisp objects (elements) and provide stack-like behavior. Each ring is limited to a maximum number of elements and can be rotated. You can control the insertion point where elements get added and removed, and iterate across the elements.

For more information about rings, start at `make-ring`.

28.5 Conditional throw and checking for catch in the dynamic environment

In some situations it may be useful to check whether there is a specific catch in the dynamic scope, and throw if there is such a catch. The function `find-throw-tag` and the macro `throw-if-tag-found` can be used in these circumstances.

28.6 Checking for a dynamic binding

Use `symbol-dynamically-bound-p` to test whether a symbol is dynamically bound in the current environment.
This chapter summarizes the technical differences between 64-bit LispWorks and 32-bit LispWorks. Both are ANSI Common Lisp implementations and support the same language extensions and libraries, so in many ways they behave the same. However the programmer should be aware of the differences mentioned here.

29.1 Introduction

64-bit LispWorks has a larger address space, subject to physical memory. The maximum heap sizes are shown in Table 29.1.

You can make larger arrays and the fixnum type is much larger than in 32-bit LispWorks. The values of various Common Lisp architectural constants reflect this, as shown in Table 29.2.

Other differences in 64-bit LispWorks are noted in the remaining sections of this chapter.
29.2 Heap size

In principle 64-bit LispWorks can grow to almost 16 TB but it is intentionally limited to a defined range in order to avoid clashes with other software as shown in Table 29.1.

Table 29.1 Default range for 64-bit LispWorks heap

<table>
<thead>
<tr>
<th>Platform</th>
<th>Default range</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel-based Macintosh</td>
<td>#x40000000000 to #x40000000000 (3.75 TB)</td>
<td></td>
</tr>
<tr>
<td>PowerPC Macintosh</td>
<td>#x40000000000 to #x40000000000 (3.75 TB)</td>
<td></td>
</tr>
<tr>
<td>old Linux</td>
<td>#x40000000000 to #x70000000000 (192 GB)</td>
<td>Effective limit around 160 GB.</td>
</tr>
<tr>
<td>modern Linux</td>
<td>#x40000000000 to #x44000000000 (4 TB)</td>
<td></td>
</tr>
<tr>
<td>Windows</td>
<td>#x40000000000 to #x40000000000 (3.75 TB)</td>
<td></td>
</tr>
<tr>
<td>Solaris</td>
<td>#x40000000000 to #x10000000000 (768 GB)</td>
<td></td>
</tr>
</tbody>
</table>

In contrast, 32-bit LispWorks has a maximum heap size of 1.5-3.0 GB depending on platform.

Normally 64-bit LispWorks for Linux automatically adjusts its default heap size on startup according to whether it runs on "old Linux" or "new Linux". On old systems, LispWorks sets the end of its range to \#x70000000000. On new systems, it sets the end to \#x44000000000, thus giving a range of 4 TB. However, if the size is given explicitly by command line argument \texttt{--reserve-size} or \texttt{InitLispWorks} in a dynamic library, then this overrides the automatic adjustment.

LispWorks is relocatable on all supported platforms as described in “Startup relocation of 32-bit LispWorks” on page 460 and “Startup relocation of 64-bit LispWorks” on page 461.
29.3 Architectural constants

Common Lisp constants have the values shown in Table 29.2

<table>
<thead>
<tr>
<th>Constant</th>
<th>32-bit LispWorks</th>
<th>64-bit LispWorks</th>
</tr>
</thead>
<tbody>
<tr>
<td>most-positive-fixnum</td>
<td>$2^{29} - 1$</td>
<td>$2^{60} - 1$</td>
</tr>
<tr>
<td>array-dimension-limit</td>
<td>67108337 (almost $2^{26}$)</td>
<td>$2^{29} - 1$</td>
</tr>
<tr>
<td>array-total-size-limit</td>
<td>$2^{26}$</td>
<td>$2^{29} - 1$</td>
</tr>
</tbody>
</table>

Note: In 32-bit LispWorks 5.0, array-total-size-limit is $2^{29} - 1$, which is wrong.

29.4 Speed

64-bit LispWorks is generally faster than 32-bit LispWorks.

We would be interested to see comparative performance data from your application if it runs on both 32-bit and 64-bit LispWorks.

29.5 Memory Management and cl:room

Memory layout and the garbage collector (GC) differ significantly between the two implementations.

For the details see “Memory Management in 32-bit LispWorks” on page 135 and “Memory Management in 64-bit LispWorks” on page 146.

The output of room differs between 64-bit and 32-bit LispWorks.

29.6 Greater allocation expected in 64-bit LispWorks

In 64-bit LispWorks pointers are 8 bytes, whereas in 32-bit LispWorks pointers are 4 bytes. Since many objects contain pointers, most programs will allocate more in 64-bit LispWorks, typically at least 50% or more.
A program containing mostly strings would not show this difference, since strings are more comparable in size between the two implementations.

You can use `find-object-size` to find the size of an object.

### 29.7 Float types

In 64-bit LispWorks `single-floats` are immediate objects, and `short-float` is the same type as `single-float`.

In 32-bit LispWorks `single-floats` are boxed objects, and `short-float` is disjoint from other float types.

### 29.8 External libraries

Third party libraries loaded into 64-bit LispWorks must be 64-bit. Availability of a suitable library is therefore a possible issue when porting your LispWorks application to 64-bit.

Third party libraries loaded into 32-bit LispWorks must be 32-bit.
This chapter enumerates the set of examples in the LispWorks library relevant to the content of this manual. Each example file contains complete, self-contained code and detailed comments, which include one or more entry points near the start of the file which you can run to start the program.

To run the example code:

1. Open the file in the Editor tool in the LispWorks IDE. Evaluating the call to `example-edit-file` shown below will achieve this.

2. Compile the example code, by `Ctrl+Shift+B`.

3. Place the cursor at the end of the entry point form and press `Ctrl+X Ctrl+E` to run it.

4. Read the comment at the top of the file, which may contain further instructions on how to interact with the example.

### 30.1 COMM examples

#### 30.1.1 SSL examples

This section lists the example files illustrating the use of SSL in socket streams, described in detail in “Using SSL” on page 418:
30.1.2 Asynchronous I/O examples
This section lists the example files illustrating the Asynchronous I/O API, described in detail in “Asynchronous I/O” on page 411:

(example-edit-file "async-io/driver")
(example-edit-file "async-io/multiplication-table")
(example-edit-file "async-io/print-connection-delay")
(example-edit-file "ssl/async-io-client")
(example-edit-file "async-io/udp")

30.2 Streams examples
(example-edit-file "streams/buffered-stream")

30.3 DDE examples
This section lists the example files illustrating Dynamic Data Exchange (DDE) on Microsoft Windows, described in detail in “Dynamic Data Exchange” on page 321:

(example-edit-file "dde/lispworks-ide")
(example-edit-file "dde/server-dispatching")
(example-edit-file "dde/server-dispatching-client")

30.4 Parser generator examples
(example-edit-file "parser-generator/expression-parser")

30.5 Examples for save-image in a Mac OS X application
bundle

This section lists the example files illustrating how you can create an application bundle while saving an image on Mac OS X. See “Saving a LispWorks image” on page 179 for details of the process:

(example-edit-file "configuration/macos-application-bundle")
(example-edit-file "configuration/save-macos-application")

Note: These examples are provided as a starting point for programmers who need to modify their own bundle-creation code. LispWorks for Macintosh has documented functions create-macos-application-bundle and save-image-with-bundle which you should use unless you need different functionality.

30.6 Miscellaneous examples

A minimal example of parsing XML:

(example-edit-file "misc/xml-parser")

Code for using ASDF in the LispWorks IDE, described in more detail in “Using ASDF” on page 311:

(example-edit-file "misc/asdf-integration")
The CLOS Package

This chapter describes the LispWorks extensions to CLOS, the Common Lisp Object System.

The LispWorks Meta Object Protocol mostly conforms to chapters 5 & 6 of AMOP. Manual pages for symbols with different functionality from AMOP are in this chapter, and the differences are discussed in Chapter 18, “The Metaobject Protocol”.

**break-new-instances-on-access**

*Function*

**Summary**

Breaks to the debugger when a new instance of a class is accessed. Note that this function is deprecated.

**Package**

clos

**Signature**

`break-new-instances-on-access class-designator &key read write slot-names when process trace-output entrycond eval-before before backtrace => t`

**Arguments**

`class-designator` The class to trap.

**Values**

Returns t.
Causes a break when new instances of the class given by class-designator are accessed, according to the keyword arguments.

The keyword arguments control which type of access cause a break and are interpreted as described for trace-on-access.

Note: this function is deprecated. You should now call trace-new-instances-on-access with :break t instead.

See also trace-new-instances-on-access

break-on-access

Summary Breaks to the debugger when an instance of a class is accessed. Note that this function is deprecated.

Package clos

Signature break-on-access instance &key read write slot-names when process trace-output entrycond eval-before before backtrace => t

Arguments instance A CLOS instance.

Values Returns t.

Description A useful debugging function which causes access to instance to break to the debugger. Accesses include calls to slot-value and also accessor functions defined by the class of instance. Other instances of the same class are unaffected.

The keyword arguments control which type of access cause a break and are interpreted as described for trace-on-access.

You can remove the break by calling unbreak-on-access.

A common use of this function is to find where a slot is being changed in a complex program.
Note: this function is deprecated. You should now call `trace-on-access` with `:break t` instead.

See also `trace-on-access`

class-extra-initargs

Generic Function

Summary
Extends the valid initialization arguments of a class.

Package
`clos`

Signature
class-extra-initargs prototype => initargs

Arguments
prototype A class prototype.

Values
initargs A list of additional initialization arguments.

Description
The generic function `class-extra-initargs` lets you extend the set of valid initialization arguments for a class and its subclasses. `initargs` should be a list of symbols. Each symbol becomes a valid initarg for the class. By default in a non-delivered LispWorks image, `make-instance` and other CLOS initializations (see `set-clos-initarg-checking`) check that initargs passed to them are valid.

The extra initargs are used for `make-instance`, `reinitialize-instance`, `update-instance-for-redefined-class` and `update-instance-for-different-class`.

Notes
`class-extra-initargs` is useful only in complex cases. In most cases other ways of extending the set of valid initargs are simpler and clearer, such as the `:extra-initargs` class option, described in `defclass`.

Example
In this session an illegal initarg `:my-keyword` is passed, causing `make-instance` to signal an error.
Then :my-keyword is added as an extra initarg, after which
make-instance accepts it.

CL-USER 38 > (defclass my-class () ((a :initform nil)))
#<STANDARD-CLASS MY-CLASS 113AAA2F>

CL-USER 39 > (make-instance 'my-class :my-keyword 8)

Error: MAKE-INSTANCE is called with unknown keyword
:MY-KEYWORD among the arguments (MY-CLASS :MY-KEYWORD
8) {no keywords allowed}
  1 (continue) Ignore the keyword :MY-KEYWORD
  2 (abort) Return to level 0.
  3 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed,
or :? for other options

CL-USER 40 : 1 > :a

CL-USER 41 > (defmethod clos:class-extra-initargs
  ((x my-class))
  '(:my-keyword))
#<STANDARD-METHOD CLOS:CLASS-EXTRA-INITARGS (MY-CLASS)
1137C763>

CL-USER 42 > (make-instance 'my-class :my-keyword 8)
#<MY-CLASS 11368963>

See also
compute-class-potential-initargs
defclass
make-instance
set-clos-initarg-checking

compute-class-potential-initargs

Generic Function

Summary Computes the valid initargs of a class.

Package clos

Signature compute-class-potential-initargs class => initargs
Arguments

- **class**: A class.

Values

- **initargs**: A list of symbols, or \( t \).

Description

The generic function `compute-class-potential-initargs` is called to compute the initialization arguments of a class. This set of valid initargs is used by `make-instance` when its arguments are checked.

`class` is the class passed to `make-instance`. That is, `compute-class-potential-initargs` specializes on the metaclass.

`initargs` is either a list of valid initargs, or \( t \) meaning that any initialization argument is allowed.

There is a supplied method on \( t \), which returns `nil`.

The other supplied method is on `standard-class`. This consults the Relevant Methods, which are the applicable methods of `make-instance`, `allocate-instance`, `initialize-instance` and `shared-initialize`. If any of the Relevant Methods have a lambda list containing `&allow-other-keys` then `initargs` is \( t \). Otherwise `initargs` is a list containing:

- all the `&key` arguments from Relevant Method lambda lists, and
- the initargs of the slots of `class` and its superclasses, and
- any extra initargs specified via the class option `:extra-initargs` (see `defclass` for details of this), and
- any extra initargs returned by `class-extra-initargs`.

The list `initargs` contains no duplicates, and the result of `compute-class-potential-initargs` is cached so that it is not recomputed unless one of the Relevant Methods, the class or its class precedence list is altered.
The CLOS Package

See also

class-extra-initargs
make-instance
set-clos-initarg-checking

**compute-discriminating-function**

Generic Function

Summary

Returns the discriminating function.

Package

clos

Signature

compute-discriminating-function gf => result

Arguments

gf A generic function.

Values

result A function.

Description

The generic function compute-discriminating-function returns the discriminator as specified in AMOP. However, there are two discrepancies with the AMOP behavior:

- The discriminating function does not compute-applicable-methods-using-classes, since this is not implemented.

- add-method does not call compute-discriminating-function. Instead, it is called when the generic function is called. This is more efficient than calling compute-discriminating-function each time add-method is called.

**compute-effective-method-function-from-classes**

Generic Function

Summary

Returns the effective method function.

Package

clos
<table>
<thead>
<tr>
<th>Signature</th>
<th>compute-effective-method-function-from-classes ( gf ) ( classes ) ( =&gt; ) ( em-function )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>( gf ) A generic function.</td>
</tr>
<tr>
<td>( classes )</td>
<td>A list of class metaobjects.</td>
</tr>
<tr>
<td>Values</td>
<td>( em-function ) A function or ( nil ).</td>
</tr>
<tr>
<td>Description</td>
<td>The generic function <code>compute-effective-method-function-from-classes</code> is called by LispWorks to compute the effective method function when ( gf ) is called with required argument types specified by ( classes ). If ( em-function ) is ( nil ), then <code>no-applicable-method</code> is called. Otherwise, <code>em-function</code> may be cached by the generic function and is called with the arguments supplied to the generic function. The default method for <code>compute-effective-method-function-from-classes</code> implements the standard generic function behavior of finding the applicable methods and using the method combination to construct a function that calls them. In order for <code>compute-effective-method-function-from-classes</code> to be called and the result cached, there must be methods specializing on the “interesting” arguments. For the standard behavior, this is trivially true, but if you want to implement other behavior then you need to define dummy methods even if they are never called.</td>
</tr>
<tr>
<td>Example</td>
<td>A &quot;computed&quot; generic function that returns a value based on a form chosen from the classes of the arguments rather than the methods. Note the dummy method which is specialized on null.</td>
</tr>
</tbody>
</table>
(defclass computed-generic-function (standard-generic-function)
  ((computer :initarg :computer
               :accessor computed-generic-function-computer))
  (:metaclass funcallable-standard-class))

(defmethod clos:compute-effective-method-function-from-classes
  ((gf computed-generic-function)
   classes)
  (apply (computed-generic-function-computer gf) gf classes))

(defmacro define-computed-generic-function (name
   lambda-list
   specializers
   &body body)
  `(dspec:def (define-computed-generic-function ,name)
    (defgeneric ,name ,lambda-list
     (:generic-function-class computed-generic-function)
     (:method ,(loop for arg in lambda-list
                     collect
                     (if (member arg specializers)
                         `(~(,arg null)
                            ,arg)))))
    (setf (computed-generic-function-computer #',name)
         #'(lambda ,name ,@(loop for arg in lambda-list
                                 collect
                                 (if (member arg specializers)
                                     arg
                                     (gensym))))
         ,@body))
  ,name))

(define-computed-generic-function aaaa (x y) (x)
  (let ((something (compute-something aaaa x)))
    #'(lambda (x y)
       (declare (ignore y))
       (format nil "Something for ~a is ~a" x something)))
  (define-computed-generic-function aaaa (x y) (x)
    (let ((something (compute-something aaaa x)))
      #'(lambda (x y)
         (declare (ignore y))
         (format nil "Something for ~a is ~a" x something)))
  )

(defun compute-something (gf class)
See also

**copy-standard-object**  
*Function*

**Summary**
Creates a new copy of a CLOS object.

**Package**
clos

**Signature**
copy-standard-object source => target

**Arguments**
source  
A standard-object, but not a funcallable-standard-object.

**Values**
target  
A standard-object, but not a funcallable-standard-object.

**Description**
The function **copy-standard-object** creates a new copy of the CLOS object source.

source must be of type standard-object, excluding funcallable-standard-object and its subclasses, in particular it cannot be of type generic-function.

The copying is shallow, that is only the actual values are copied, as if by

(dolist (slot instance-slots)  
  (setf (slot-value target slot)  
    (slot-value source slot)))

assuming no definition that affects what slot-value and (setf slot-value) do. However, **copy-standard-object** bypasses the slot-value mechanism and is much faster.

**copy-standard-object** should be used on instances of user-defined classes which do not inherit from system-defined
classes (other than standard-object). If source is an instance of a system-defined class (or a subclass of a system-defined class) then target cannot be used as a functional object, but its slot values can be read safely. That may be useful for debugging.

See also replace-standard-object

**funcallable-standard-object**

*Class*

**Package** clos

**Superclasses** function

standard-object

**Subclasses** generic-function

**Description**
The metaclass funcallable-standard-object provides the default :direct-superclasses for instances of funcallable-standard-class and its subclasses.

funcallable-standard-object is implemented as described in AMOP except for a different order in the class precedence list.

In AMOP the class precedence list is

(funcallable-standard-object standard-object function t)

whereas in LispWorks the class precedence list is

(funcallable-standard-object function standard-object t)

LispWorks is like this to be compliant with the rules in the ANSI Common Lisp Standard.

The AMOP class precedence list implies a class precedence for generic-function which violates the last sentence in
Generic Function

**process-a-class-option**

**Summary**
Describes how the value of a class option is parsed.

**Package**
clos

**Signature**
`process-a-class-option metaclass option value => initargs`

**Arguments**
- `metaclass`: The metaclass of the class being parsed.
- `option`: The `defclass` option name.
- `value`: The tail of the `defclass` option form.

**Values**
- `initargs`: A plist of initargs describing the option.

**Description**
The generic function `process-a-class-option` describes how the value of a class option is parsed. It is called at `defclass` macroexpansion time. By default, LispWorks parses class options as defined in AMOP, but you need to supply a method if you need class options with different behavior.

`initargs` should be a plist of class initargs and values. These are added to any other initargs for the class.

**Example**

```
(defclass m1 (standard-class)
  ((title :initarg :title)))
```

For single-valued, evaluated title option, add a method like this:
(defmethod clos:process-a-class-option
  ((class m1)
   (name (eql :title))
   value)
  (unless (and value (null (cdr value)))
   (error "m1 :title must have a single value."))
  (list name (car value)))

(defclass my-titled-class ()
  ()
  (:metaclass m1)
  (:title "Initial Title"))

If the value is not to be evaluated, the method would look like this:

(defmethod clos:process-a-class-option
  ((class m1)
   (name (eql :title))
   value)
  (unless (and value (null (cdr value)))
   (error "m1 :title must have a single value."))
  (list name ,value))

Now suppose we want an option whose value is a list of titles:

(defclass m2 (standard-class)
  ((titles-list :initarg :list-of-possible-titles)))

If the titles are to be evaluated, add a method like this:

(defmethod clos:process-a-class-option
  ((class m2)
   (name (eql :list-of-possible-titles))
   value)
  (list name (list ,@value)))

Or, if the titles should not be evaluated, add a method like this:
(defmethod clos:process-a-class-option
  ((class m2)
   (name (eql :list-of-possible-titles))
   value)
  (list name `',value))

(defclass my-multi-titled-class ()
  ()
  (:metaclass m2)
  (:list-of-possible-titles
   "Initial Title 1"
   "Initial Title 2"))

See also defclass
process-a-slot-option

process-a-slot-option Generic Function

Summary Describes how a defclass slot option is parsed.

Package clos

Signature process-a-slot-option metaclass option value already-processed-other-options slot => processed-options

Arguments

  metaclass The metaclass of the class being parsed.
  option The slot option name.
  value The value of the slot option.

  already-processed-other-options
  A plist of initargs for non-standard options that have been processed already.

  slot The whole slot description.

Values processed-options

A plist of initargs.
Description

The generic function \texttt{process-a-slot-option} describes how the value of a slot option is parsed. It is called at \texttt{defclass} macroexpansion time. By default LispWorks parses slot options as defined in AMOP, but you need to supply a method if you need slot options with different behavior.

\texttt{processed-options} should be a plist of slot initargs and values containing those from \texttt{already-processed-other-options} together with initargs for \texttt{option} as required. These are added to any other initargs for the slot.

Example

\begin{verbatim}
(defun extended-class (standard-class) ()
(defmethod clos:process-a-slot-option
  ((class extended-class) option value
   already-processed-options slot)
  (if (eq option :extended-slot)
      (list* :extended-slot
             value
             already-processed-options)
      (call-next-method)))

(defclass extended-direct-slot-definition
  (clos:standard-direct-slot-definition)
  ((extended-slot :initarg :extended-slot :initform nil)))

(defmethod clos:direct-slot-definition-class
  ((x extended-class) &rest initargs)
  'extended-direct-slot-definition)

(defclass test ()
  ((regular :initform 3)
   (extended :extended-slot t :initform 4))
  (:metaclass extended-class))
\end{verbatim}

To add a slot option \texttt{:special-reader} whose value is a non-evaluated symbol naming a reader:
(defmethod clos:process-a-slot-option
    ((class my-metaclass) option value
     already-processed-options slot)
  (if (and (eq option :special-reader)
     (symbolp value))
    (list* :special-reader`
      ,value
     already-processed-options)
    (call-next-method)))

To allow repeated \texttt{special-reader} options which are combined into a list:

(defmethod clos:process-a-slot-option
    ((class my-metaclass) option value
     already-processed-options slot)
  (if (and (eq option :special-reader) (symbolp value))
    (let ((existing (getf
      already-processed-options
      :special-reader)))
      (if existing ; this is a quoted list of symbols
        (progn
          (setf (cdr (last (cadr existing))) (list
            value))
        already-processed-options)
        (list* :special-reader`
          ,(value)
        already-processed-options)))
    (call-next-method)))

See also \texttt{defclass}

\texttt{process-a-class-option}

\section*{replace-standard-object}

\textbf{Function}

\textbf{Summary} Replaces the values in a CLOS object's slots by the values of slots from another object.

\textbf{Package} clos

\textbf{Signature} replace-standard-object target source => target
Arguments

source, target
A standard-object, but not a funcallable-standard-object.

Description

The function replace-standard-object replaces the values in the slots of the CLOS object target by the values of slots from the CLOS object source.

Only slots with allocation type :instance are copied from source to target.

source and target must be of type standard-object, excluding funcallable-standard-object and its subclasses, in particular they cannot be of type generic-function. Moreover both must be of the same class, that is:

(eq (class-of target) (class-of source)) => t

The replacement is shallow, that is only the actual values are copied, as if by

(dolist (slot instance-slots)
  (setf (slot-value target slot)
        (slot-value source slot)))

assuming no definition that affects what slot-value and (setf slot-value) do. However, replace-standard-object bypasses the slot-value mechanism and is much faster.

replace-standard-object should be used on instances of user-defined classes which do not inherit from system-defined classes (other than standard-object). It should never be used on instances of system-defined classes and their subclasses.

See also

copy-standard-object
set-clos-initarg-checking  

**Function**

**Summary**
Switches initarg checking on or off in *make-instance*, *reinitialize-instance*, *change-class* and so on.

**Package**
clos

**Signature**
set-clos-initarg-checking *on* => *on*

**Arguments**
on  A generalized boolean.

**Description**
The function `set-clos-initarg-checking` provides control over whether CLOS checks initialization arguments. Initializations affected include:

- Calls to `make-instance`
- Calls to `reinitialize-instance`
- Calls to `change-class`
- `call-next-method` to `update-instance-for-redefined-class` with extra keywords.

Calling `set-clos-initarg-checking` with a true value of *on* causes the above initializations to check their initargs. This is the initial state of LispWorks.

Initarg checking is switched off globally and dynamically by

```lisp
(set-clos-initarg-checking nil)
```

**Notes**

1. The effect of calling `set-clos-initarg-checking` can be overridden in a runtime by the `deliver` keyword argument `:clos-initarg-checking`. See the *LispWorks Delivery User Guide* for details.


**See also**
class-extra-initargs  
compute-class-potential-initargs
deliver
make-instance

set-make-instance-argument-checking  Function

Summary
Switches CLOS initarg checking on or off. This function is deprecated.

Package  clos

Signature  set-make-instance-initarg-checking on => on

Arguments  on  A boolean.

Description
The deprecated function set-make-instance-initarg-checking switches CLOS initarg checking on or off.

Notes
set-make-instance-initarg-checking is deprecated. It is an alias for set-clos-initarg-checking.

Compatibility notes
1. In LispWorks 6.1 and later versions set-make-instance-initarg-checking affects CLOS initializations other than make-instance. For clarity, you should now use set-clos-initarg-checking instead.

2. In LispWorks 6.0 set-make-instance-initarg-checking affects only make-instance.

See also
set-clos-initarg-checking

slot-boundp-using-class  Generic Function

Summary  Implements slot-boundp.

Package  clos
Signature  
slot-boundp-using-class class object slot-name => result

Arguments  
class A class metaobject, the class of object.
object An object.
slot-name A slot name.

Values  
result A boolean.

Description  
The generic function slot-boundp-using-class implements the behavior of the slot-boundp function.

The implementation is as described in AMOP, except that the third argument is the slot name, and not a slot definition metaobject. The primary methods specialize on t for this argument.

See also  
slot-makunbound-using-class
slot-value-using-class

slot-makunbound-using-class  
Generic Function

Summary  
Implements slot-makunbound.

Package  
clos

Signature  
slot-makunbound-using-class class object slot-name => object

Arguments  
class A class metaobject, the class of object.
object An object.
slot-name A slot name.

Values  
object The object argument.

Description  
The generic function slot-makunbound-using-class implements the behavior of the slot-makunbound function.
The implementation is as described in AMOP, except that the third argument is the slot name, and not a slot definition metaobject. The primary methods specialize on t for this argument.

See also

slot-boundp-using-class
slot-value-using-class

---

**slot-value-using-class**

*Generic Function*

**Summary**

Implements `slot-value`.

**Package**

`clos`

**Signature**

```
(slot-value-using-class class object slot-name => value)
(setf slot-value-using-class) value class object slot-name => value
```

**Arguments**

- `class`  
  A class metaobject, the class of `object`.
- `object`  
  An object.
- `slot-name`  
  A slot name.

**Values**

- `value`  
  The value of the slot named by `slot-name`.

**Description**

The generic function `slot-value-using-class` implements the behavior of the `slot-value` function.

The implementation is as described in AMOP, except that the third argument is the slot name, and not a slot definition metaobject. The primary methods specialize on t for this argument.

**Note:** by default, standard slot accessors are optimized to not call `slot-value-using-class`. This can be overridden with the `:optimize-slot-access` class option. See `defclass` for details.
See also  
defclass  
slot-boundp-using-class  
slot-makunbound-using-class

**trace-new-instances-on-access**

**Function**

**Summary**  
Traces new instances of a given class, based on access modes.

**Package**  
clos

**Signature**  

```
trace-new-instances-on-access class-designator
&key read write slot-names break when process trace-output
entrycond eval-before before backtrace => t
```

**Arguments**  

- `class-designator`  
The class to trace.

**Values**  

Returns t.

**Description**  
Causes new instances of the class given by `class-designator` to be traced for the access modes given by `read`, `write` and `slot-names`.

The keyword arguments control which type of access are traced, and provide preconditions for tracing, code to run before access, and how to print any trace output. They are interpreted as described for `trace-on-access`.

This function, when used with the `:break` keyword, replaces the deprecated function `break-new-instances-on-access`.

**Example**  

```
(trace-new-instances-on-access 'capi:display-pane
   :slot-names nil)
```

Suppose you have a bug whereby the slot `bar` of an instance of your class `foo` is incorrectly being set to a negative integer value. You could cause entry into the debugger at the point where the slot is set incorrectly by evaluating this form:
(clos:trace-new-instances-on-access
 'foo
 :slot-names '(bar)
 :read nil
 :when '(and (integerp (car *traced-arglist*))
 (<= (car *traced-arglist*) 0))
 :break t)

and running your program.

See also break-new-instances-on-access
untrace-new-instances-on-access
trace-on-access

trace-on-access

Function

Summary Invokes the trace facilities when an instance of a class is accessed.

Package clos

Signature trace-on-access instance &key read write slot-names break when process trace-output entrycond eval-before before backtrace => t

Arguments instance A CLOS instance.
read A generalized boolean.
write A generalized boolean.
slot-names A list of symbols, or t.
break A generalized boolean.
when A form.
process A form.
trace-output A form.
entrycond A form.
eval-before A list of forms.
before A list of forms.

backtrace A keyword, t or nil.

Values

<table>
<thead>
<tr>
<th>Properties</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns t.</td>
</tr>
</tbody>
</table>

Description

A useful debugging function which causes access to instance to invoke the trace facilities. Accesses include calls to slot-value and accessor functions defined by the class of instance.

The keyword arguments control which type of access are traced, and provide preconditions for tracing, code to run before access, and how to print any trace output. They are similar to those supported by the trace macro (but note that these CLOS symbols are functions, so the keyword values are evaluated immediately, unlike in trace).

read controls whether reading slots is traced. The default is t.

write controls whether writing slots is traced. The default is t.

slot-names controls which slots to trace access for. It can be a list of symbols which are the slot-names. The default value, t, means trace access to all slots.

break controls whether the debugger is entered when a traced slot in instance is accessed. When nil, the debugger is not invoked and messages are printed to *trace-output*. The default value is nil.

when is evaluated during slot access to determine whether any tracing should occur. The default value is t.

process is evaluated during slot access to determine whether any tracing should occur in the current process. The form should evaluate to either nil (meaning trace in all processes), a string naming the process in which tracing should occur (see process-name, find-process-from-name), or a list of strings naming the processes in which tracing should occur. The default value is nil.
trace-output is evaluated during slot access to determine the stream on which to print tracing messages. If this is nil then the value of *trace-output* is used. The default value is nil.

entrycond is evaluated during slot access to determine whether the default tracing messages should be printed.

eval-before is a list of forms which are evaluated during slot access.

before is a list of forms which are evaluated during slot access. The first value returned by each form is printed.

backtrace controls what kind of backtrace to print. If this is nil then no backtrace is printed, and this is the default value. Otherwise it can be any of the following values:

:quick Like the :bq debugger command.

t Like the :b debugger command.

:verbose Like the :b :verbose debugger command.

:bug-form Like the :bug-form debugger command.

Other instances of the same class are unaffected and you can remove the trace by calling untrace-on-access.

The variable *traced-arglist* is bound to a list of arguments for the slot access during evaluation of the options above, that is (instance slot-name) when reading a slot and (new-value instance slot-name) when writing a slot.

A common use of this function is to find where a slot is being changed in a complex program.

This function, when called with :break t, replaces the deprecated function break-on-access.

See also untrace-on-access
trace-new-instances-on-access
break-on-access
unbreak-new-instances-on-access  

**Summary**  
Removes the trapping installed by `break-new-instances-on-access`. Note that this function is deprecated.

**Package**  
clos

**Signature**  
`unbreak-new-instances-on-access class-designator => t`

**Arguments**  
`class-designator`  
The class whose trap you want to remove.

**Values**  
Returns `t`.

**Description**  
Removes the trapping installed by `break-new-instances-on-access`. Note that this function is deprecated. You should now use `untrace-new-instances-on-access` instead.

**See also**  
`untrace-new-instances-on-access`

unbreak-on-access  

**Summary**  
Removes the trapping installed by `break-on-access`. Note that this function is deprecated.

**Package**  
clos

**Signature**  
`unbreak-on-access instance`

**Arguments**  
`instance`  
A class instance

**Values**  
Returns `t`.

**Description**  
Removes any break installed on `instance` by `break-on-access`. See `untrace-on-access` for details.

**Note:** this function is deprecated. You should now use `untrace-on-access` instead.
See also  

untrace-new-instances-on-access

Function

Summary  
Removes the tracing installed by `trace-new-instances-on-access`.

Package  
clos

Signature  
`untrace-new-instances-on-access class-designator => t`

Arguments  
`class-designator`  
The class whose trap you want to remove.

Values  
Returns t.

Description  
Removes the tracing installed by `trace-new-instances-on-access`.

See also  
`trace-new-instances-on-access`  
`untrace-on-access`

untrace-on-access

Function

Summary  
Removes the tracing installed by `trace-on-access`.

Package  
clos

Signature  
`untrace-on-access instance => t`

Arguments  
`instance`  
A class instance

Values  
Returns t.

Description  
Removes any trace installed on `instance` by `trace-on-access`. 
See also

trace-on-access
untrace-new-instances-on-access
The CLOS Package
The COMM Package

This chapter provides reference entries for the functions in the COMM package.

The COMM package provides the TCP/IP interface. TCP/IP sockets can be used to communicate between processes and machines and the mechanism allows LispWorks to connect to or implement a server. It also allows using Secure Sockets Layer (SSL) processing in the socket.

The COMM package also provides the Asynchronous I/O API including UDP sockets as described in “Asynchronous I/O” on page 411.

An overview of this functionality is in Chapter 25, “TCP and UDP socket communication and SSL”.

Before the interface can be used the module "comm" must be loaded using

```
(require "comm")
```

accept-tcp-connections-creating-async-io-states

Function

Summary
Starts accepting TCP connections to a port within a wait-state-collection.

Package
comm
Signature

accept-tcp-connections-creating-async-io-states
collection service connection-function &key init-function init-timeout
backlog address nodelay keepalive ipv6 create-state name queue-output
handle-name user-info ssl-ctx ssl-side ctx-configure-callback
ssl-configure-callback handshake-timeout ssl-error-callback =>
accepting-handle

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collection</td>
<td>A <code>wait-state-collection</code>.</td>
</tr>
<tr>
<td>service</td>
<td>An integer, a string or <code>nil</code>.</td>
</tr>
<tr>
<td>connection-function</td>
<td>A function designator.</td>
</tr>
<tr>
<td>init-function</td>
<td><code>nil</code> or a function designator.</td>
</tr>
<tr>
<td>init-timeout</td>
<td><code>nil</code> or a non-negative real number.</td>
</tr>
<tr>
<td>backlog</td>
<td><code>nil</code> or a positive integer.</td>
</tr>
<tr>
<td>address</td>
<td>An integer, an <code>ipv6-address</code> object, a string or <code>nil</code>.</td>
</tr>
<tr>
<td>nodelay</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>keepalive</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>ipv6</td>
<td>The keyword <code>:any</code>, <code>nil</code>, <code>t</code> or the keyword <code>:both</code>.</td>
</tr>
<tr>
<td>create-state</td>
<td>A boolean.</td>
</tr>
<tr>
<td>queue-output</td>
<td>A boolean.</td>
</tr>
<tr>
<td>name</td>
<td>A Lisp object.</td>
</tr>
<tr>
<td>handle-name</td>
<td>A Lisp object.</td>
</tr>
<tr>
<td>user-info</td>
<td>A Lisp object.</td>
</tr>
<tr>
<td>ssl-side</td>
<td>One of the keywords <code>:client</code>, <code>:server</code> or <code>:both</code>.</td>
</tr>
<tr>
<td>ssl-ctx</td>
<td>A symbol or a foreign pointer.</td>
</tr>
</tbody>
</table>
ctx-configure-callback
A function designator or nil. The default value is nil.

ssl-configure-callback
A function designator or nil. The default value is nil.

handshake-timeout
A real or nil (the default).

ssl-error-callback
A function designator..

Values
  accepting-handle  An accepting-handle object.

Description
The function accept-tcp-connections-creating-async-io-states starts accepting TCP connections to the port service within the wait-state-collection collection.

service is interpreted as described in "Specifying the target for connecting and binding a socket" on page 408.

Each time a connection is made, connection-function is called with two arguments: accepting-handle and (by default) a new async-io-state for the connected socket. The function typically calls async-io-state-read-buffer, async-io-state-write-buffer or async-io-state-read-with-checking to start performing I/O. The keyword :create-state can be used to tell accept-tcp-connections-creating-async-io-states not to create the state and instead pass the socket itself. This is useful when you want to do the I/O "somewhere else", either by creating a socket-stream and using ordinary read/write functions on it, or using a different wait-state-collection. The default value of create-state is t.
If `init-function` is non-nil, it is called after the listening socket has been bound to the service. `init-function` should take one argument: `socket`. `socket` is the socket used by the server, which can be used to determine the bound port number by calling `get-socket-address`.

If the port number specified by `service` is already in use, then `accept-tcp-connections-creating-async-io-states` periodically tries to bind to the port number for up to 1 minute (or `init-timeout` seconds if this is non-nil).

`queue-output` controls what happens if you try to perform a write operation on any of the states that `accept-tcp-connections-creating-async-io-states` creates while another write operation is in progress on the same state. When `queue-output` is `nil`, such an operation will cause an error. When `queue-output` is non-nil, the second write operation is queued and actually executed later. The default value of `queue-output` is `nil`.

The result `accepting-handle` is an object of type `accepting-handle`, which is the same object that will be passed to the `connection-function`. It can be used to stop accepting and closing the socket by `close-accepting-handle`, and also retrieving the socket.

`handle-name` and `user-info` are stored in the `accepting-handle` object. The `user-info` is not touched in any way by the system, and it is intended for you to pass information to the `connection-function`. `handle-name` is used when printing the handle, but is not accessed otherwise.

When `ssl-ctx` is non-nil, `accept-tcp-connections-creating-async-io-states` always creates an `async-io-state` (ignoring `create-state`), and attaches SSL to it. `ssl-side`, `ssl-ctx`, `ctx-configure-callback`, `ssl-configure-callback` and `handshake-timeout` are interpreted as described in “Keyword arguments for use with SSL” on page 421.
ssl-error-callback defaults to error. It is called when there is any error while attaching SSL to the new socket. Such errors can be either the result of an error in the configuration functions, or (more commonly) an error during the SSL handshake. ssl-error-callback is called from the thread of the wait-state-collection that was passed to accept-tcp-connections-creating-async-io-states. The socket is discarded before Sssl-error-callback is called.

For details of backlog, address, nodelay, keepalive and ipv6, see start-up-server.

The default value of nodelay is t.

The default value of ipv6 is :any.

The default value of name is a string "Listening".

Notes

accept-tcp-connections-creating-async-io-states binds the socket synchronously, that is when it returns successfully the socket is already bound. However, it already started accepting connections. If you need to access the socket after binding and before starting to accept connections, use the init-function argument.

See also

create-async-io-state
create-async-io-state-and-connected-tcp-socket
“The Async-I/O-State API” on page 412
accepting-handle
accepting-handle-local-port
close-accepting-handle

accepting-handle

Type

Summary

The type of object returned by accept-tcp-connections-creating-async-io-states.

Package

comm
accepting-handle is the type of the object that is returned by `accept-tcp-connections-creating-async-io-states` and is passed as the first argument to the `connection-function` of `accept-tcp-connections-creating-async-io-states`.

The handle contains the collection with which it is associated, the underlying socket, and the `user-info` and `handle-name` that were passed to `accept-tcp-connections-creating-async-io-states`.

See also:
- `accept-tcp-connections-creating-async-io-states`
- `close-accepting-handle`
- `accepting-handle-socket`
- `accepting-handle-collection`
- `accepting-handle-local-port`
- `accepting-handle-user-info`
- `accepting-handle-name`

---

**accepting-handle-collection**

**Function**

**Summary**

Returns the collection associated with an accepting handle.

**Package**

`comm`

**Signature**

```
accepting-handle-collection accepting-handle => result
```

**Arguments**

`accepting-handle` An accepting-handle.

**Values**

`result` A collection or nil.

**Description**

The function `accepting-handle-collection` returns the collection associated with `accepting-handle`. 
accepting-handle has to be an accepting handle, currently that means the result of accept-tcp-connections-creating-async-io-states.

result is the collection that was supplied to accept-tcp-connections-creating-async-io-states, but for a closed handle result is nil.

See also

accepting-handle
accept-tcp-connections-creating-async-io-states

accepting-handle-name

Function

Summary
Returns the name associated with an accepting handle.

Package
comm

Signature
accepting-handle-name accepting-handle => name

Arguments
accepting-handle An accepting-handle.

Values
name A Lisp object.

Description
The function accepting-handle-name returns the name associated with the accepting handle accepting-handle, which is the handle-name argument to accept-tcp-connections-creating-async-io-states.

This name is used when printing the handle, so its printed representation should be reasonably short. Otherwise it is not restricted.

See also

accepting-handle
accept-tcp-connections-creating-async-io-states
accepting-handle-local-port

Function

Summary
Returns the local port number to which the socket in an accepting-handle was bound.

Package comm

Signature accepting-handle-local-port accepting-handle => port-number

Arguments accepting-handle An object of type accepting-handle.

Values port-number An integer.

Description The function accepting-handle-local-port returns the local port number to which the socket in the accepting-handle was bound.

See also accepting-handle accept-tcp-connections-creating-async-io-states

accepting-handle-socket

Function

Summary Returns the socket associated with an accepting handle.

Package comm

Signature accepting-handle-socket accepting-handle => result

Arguments accepting-handle An accepting-handle.

Values result A socket or nil.

Description The function accepting-handle-socket returns the socket associated with accepting-handle.
accepting-handle has to be an accepting handle, currently that means the result of accept-tcp-connections-creating-async-io-states.

result is the socket that was created by accept-tcp-connections-creating-async-io-states, but for a closed handle result is nil.

Notes
The socket "belongs" to the handle, and cannot be used for communication by other code. You can use accessors like get-socket-address on it.

See also
accepting-handle
accept-tcp-connections-creating-async-io-states

accepting-handle-user-info

Function

Summary
Returns the user-info associated with an accepting handle.

Package
comm

Signature
accepting-handle-user-info accepting-handle => result

Arguments
accepting-handle An accepting-handle.

Values
result A Lisp object.

Description
The function accepting-handle-user-info returns the user-info associated with the handle accepting-handle, which is the user-info argument to accept-tcp-connections-creating-async-io-states.

The system does nothing with the user-info, and its purpose is to allow you to pass information to the connection-function of accept-tcp-connections-creating-async-io-states.
See also
accepting-handle
accept-tcp-connections-creating-async-io-states

**apply-in-wait-state-collection-process**

*Function*

**Summary**
Applies a function in the process that is associated with a wait-state-collection.

**Package**
comm

**Signature**
apply-in-wait-state-collection-process collection function &rest args

**Arguments**
collection A wait-state-collection.
function A function designator.
args Lisp objects.

**Description**
The function apply-in-wait-state-collection-process applies function to the arguments args in the process that is associated with collection.

A process is associated with collection when it calls wait-for-wait-state-collection, typically from loop-processing-wait-state-collection. Normally only one process will do this for each individual wait-state-collection.

apply-in-wait-state-collection-process is asynchronous. It sends an appropriate message to the process or collection, and returns immediately, even if it is called from that process. The application happens at an undefined time inside the scope of call to call-wait-state-collection for collection.

There is no documented return value.
See also  
- call-wait-state-collection
- loop-processing-wait-state-collection
- wait-state-collection

**async-io-state**  
*System Class*

**Summary**
A class of objects that can be used to perform asynchronous I/O.

**Package**
comm

**Description**
Instances of the system class `async-io-state` can be used to perform asynchronous I/O.

See also  
- async-io-state-collection
- async-io-state-max-read
- async-io-state-name
- async-io-state-object
- async-io-state-old-length
- async-io-state-read-status
- async-io-state-read-timeout
- async-io-state-write-status
- async-io-state-write-timeout
- async-io-state-user-info
- “The Async-I/O-State API” on page 412

**async-io-state-abort**  
*Function*

**Summary**
Stops I/O and callbacks on an `async-io-state` and calls an abort callback.

**Package**
comm

**Signature**
`async-io-state-abort async-io-state abort-callback &optional direction`
Arguments

- **async-io-state**: An `async-io-state`.
- **abort-callback**: A function designator.
- **direction**: One of the keywords `:input`, `:output` and `:io`.

Values

None.

Description

The function `async-io-state-abort` stops further I/O and calls to any callbacks for direction `direction` in `async-io-state` and asynchronously calls `abort-callback` with the same arguments that the callback for a running operation would be called, except when `direction` is `:io`, when the callback is called with the state only.

The default value of `direction` is `:input`.

If by the time `abort-callback` is called there is no active operation, then `abort-callback` is called with `async-io-state` and `nil` for the other arguments.

If `async-io-state-abort` is called while a callback is running, its effect is delayed until the callback returns.

`abort-callback` can do what the other callbacks can do. In particular, it can reuse `async-io-state`, and when it aborts `async-io-state-read-with-checking` it can decide how much of the buffered data to discard by calling `async-io-state-discard`.

Notes

Due to the asynchronous delay between the time that `async-io-state-abort` is called and the time that `abort-callback` is called, the callback of the operation may have already been called, so if the `abort-callback` does anything except closing `async-io-state` it will normally have to check the state's `async-io-state-read-status`.

See also

- `async-io-state-abort-and-close`  
  “The Async-I/O-State API” on page 412
async-io-state-abort-and-close  

**Function**

**Summary**
Aborts any I/O on an **async-io-state**, closes it and optionally calls a callback.

**Package**
comm

**Signature**

**Arguments**
- `async-io-state`  
  An **async-io-state**.
- `keep-alive-p`  
  A generalized boolean.
- `close-callback`  
  A function designator for a function of one argument, or **nil**.

**Values**
None.

**Description**
The function **async-io-state-abort-and-close** aborts any I/O on **async-io-state**, closes it and optionally calls the `close-callback`.

**async-io-state-abort-and-close** first aborts any I/O operation that is in progress, and then closes the state (using **close-async-io-state**).

The value of `keep-alive-p` is passed to **close-async-io-state**.

If `close-callback` is non-nil, it should be a function taking one argument. It is called with the state as its argument after the state is closed.

**async-io-state-abort-and-close** is asynchronous. The state is known to be closed only when the `close-callback` is called.

See **close-async-io-state** about accessing the state after it is closed.
See also async-io-state-abort
   close-async-io-state
   “The Async-I/O-State API” on page 412

async-io-state-address

Summary    Returns the local address and port number for an async-io-state that has a socket.

Package    comm

Signature   async-io-state-address async-io-state => address, port


Values      address          An integer or an ipv6-address object.
            port              An integer.

Description  The function async-io-state-address returns the local address and port number for async-io-state if it has a socket (currently all states).

            address is the local host address of the socket in the state.
            port is the local port number of the socket in the state.

See also    async-io-state-peer-address
            get-socket-address
            “The Async-I/O-State API” on page 412

async-io-state-attach-ssl

Summary    Attaches SSL to an async-io-state that contains a TCP socket.

Package    comm
**Signature**

```
```

**Arguments**

- **async-io-state**: An async-io-state.
- **callback**: A function designator for a function with two arguments.
- **ssl-side**: One of the keywords :client, :server or :both.
- **ssl-ctx**: A symbol or a foreign pointer.
- **ctx-configure-callback**: A function designator or nil. The default value is nil.
- **ssl-configure-callback**: A function designator or nil. The default value is nil.
- **handshake-timeout**: A real or nil (the default).

**Values**

None

**Description**

The function `async-io-state-attach-ssl` attaches SSL to `async-io-state`, which must contain a TCP socket, typically the result of `create-async-io-state` or a state created by `accept-tcp-connections-creating-async-io-states`. `async-io-state` must not have SSL attached to it already.

`ssl-side`, `ssl-ctx`, `ctx-configure-callback`, `ssl-configure-callback` and `handshake-timeout` are interpreted as described in “Keyword arguments for use with SSL” on page 421. `ssl-ctx` defaults to t and `ssl-side` defaults to :server.

When SSL has been attached successfully or otherwise, `callback` is called with two arguments: `async-io-state` and an error-indicator. The error-indicator is nil when successful, other-
wise it is a list of a format control-string and args, suitable for applying to \texttt{format}. When the error-indicator is non-nil, \texttt{async-io-state} is not attached to SSL.

\texttt{async-io-state-attach-ssl} must not be called when there is any other operation on \texttt{async-io-state} and new operations on \texttt{async-io-state} must not be started until \texttt{callback} has been called.

\textbf{Notes}

create-async-io-state-and-connected-tcp-socket and accept-tcp-connections-creating-async-io-states can attach SSL themselves, and in most cases that is the best way to do it. \texttt{async-io-state-attach-ssl} allows the attachment to be done later.

\textbf{See also}


“Using Asynchronous I/O with SSL” on page 421

\textbf{async-io-state-buffered-data-length} \hfill \textit{Function}

\textbf{Summary}

Returns the length of the buffered data in an \texttt{async-io-state}.

\textbf{Package}

\texttt{comm}

\textbf{Signature}

\texttt{async-io-state-buffered-data-length \async-\textit{io-state} \rightarrow length}

\textbf{Arguments}

\texttt{async-\textit{io-state}} \hspace{1em} An \texttt{async-\textit{io-state}}.

\textbf{Values}

\texttt{length} \hspace{1em} A non-negative integer.

\textbf{Description}

The function \texttt{async-\textit{io-state-buffered-data-length}} returns the length of the buffered data in \texttt{async-\textit{io-state}}.
See also async-io-state-get-buffered-data
“The Async-I/O-State API” on page 412

async-io-state-ctx

Function

Summary
Accesses the SSL_CTX attached to an async-io-state.

Package
comm

Signature
async-io-state-ctx async-io-state => ssl-ctx-pointer

Arguments

Values
ssl-ctx-pointer  A foreign pointer of type ssl-ctx-pointer, or nil.

Description
The function async-io-state-ctx accesses the SSL_CTX that is attached to the async-io-state async-io-state.
It returns nil if SSL is not attached.

See also
async-io-state
ssl-ctx-pointer

async-io-state-detach-ssl

Function

Summary
Detaches SSL to an async-io-state that contains a TCP socket.

Package
comm

Signature
async-io-state-detach-ssl async-io-state callback

Arguments
callback  A function designator for a function with one argument.

Values  None

Description  The function \texttt{async-io-state-detch-ssl} detaches SSL from \texttt{async-io-state}. Subsequent communications through \texttt{async-io-state} will be without SSL.

When SSL has been detached, \texttt{callback} is called with \texttt{async-io-state}.

If \texttt{async-io-state} did not have SSL attached then \texttt{async-io-state-detch-ssl} has no effect.

\texttt{async-io-state-detch-ssl} must not be called when there is any other operation on \texttt{async-io-state}, and new operations on \texttt{async-io-state} must not be started until \texttt{callback} has been called.

Notes  There is no need to call \texttt{async-io-state-detch-ssl} before \texttt{close-async-io-state} because that also detaches SSL.

See also  \texttt{async-io-state-attach-ssl}
\texttt{close-async-io-state}
“Using Asynchronous I/O with SSL” on page 421

\texttt{async-io-state-collection}  \hspace{1cm} Function

Summary  Returns the \texttt{wait-state-collection} of an \texttt{async-io-state}.

Package  \texttt{comm}

Signature  \texttt{async-io-state-collection async-io-state => collection}

Arguments  \texttt{async-io-state}  An \texttt{async-io-state}. 
Values

collection A wait-state-collection.

Description

See also
async-io-state
wait-state-collection
“The Async-I/O-State API” on page 412

async-io-state-discard

Summary
Discards some bytes from the internal buffer in an async-io-state.

Package
comm

Signature
async-io-state-discard async-io-state discard => unread-buffer-length

Arguments
discard A positive integer.

Values
unread-buffer-length A non-negative integer.

Description
The function async-io-state-discard discards the first discard bytes from the internal buffer in async-io-state. The rest of the buffer is preserved for future reading.

async-io-state-discard must only be called inside the scope of the callback of async-io-state-read-with-checking. Once async-io-state-discard has been called, the callback must not access the buffer again.

The return value unread-buffer-length is the remaining number of bytes in the buffer.
See also  
async-io-state-read-with-checking
“The Async-I/O-State API” on page 412

**async-io-state-finish**  
*Function*

**Summary**
Stops the current read operation in an `async-io-state`.

**Package**
comm

**Signature**
`async-io-state-finish async-io-state &optional discard => unread-buffer-length`

**Arguments**
- `discard` A positive integer or `nil`.

**Values**
- `unread-buffer-length` A non-negative integer.

**Description**
The function `async-io-state-finish` stops the current read operation in `async-io-state`, so no further calls to the `async-io-state-read-with-checking callback` will occur. If `discard` is non-nil, then it discards the first `discard` bytes from the internal buffer in `async-io-state`. The rest of the buffer is preserved for future reading.

`async-io-state-finish` must only be called inside the scope of the `callback` of `async-io-state-read-with-checking`. Once `async-io-state-finish` has been called, the `callback` must not access the buffer again and a new read operation can be started.

The return value `unread-buffer-length` is the remaining number of bytes in the buffer.

See also  
async-io-state-read-with-checking
“The Async-I/O-State API” on page 412
async-io-state-get-buffered-data

Function

Summary
Copies buffered data from an async-io-state and discards it from the state.

Package
comm

Signature
async-io-state-get-buffered-data async-io-state buffer &key
start end => length

Arguments
buffer A cl:base-string or an 8-bit cl:simple-array.
start A lower bounding index designator for buffer.
end An upper bounding index designator for buffer.

Values
length A non-negative integer.

Description
The function async-io-state-get-buffered-data copies to the buffer buffer (between start and end) as much as possible of the buffered data in async-io-state and discards it from async-io-state.

The default value of start is 0. The default value of end is the length of buffer.

The return value length is the number of elements copied into buffer.

async-io-state-get-buffered-data cannot be called while an operation is active in async-io-state, that is between the call to async-io-state-read-buffer or async-io-state-write-buffer, and the call to the callback or async-io-state-abort, or between a call to async-io-state-read-with-checking and the call to async-io-state-finish or async-io-state-abort.
Use **async-io-state-buffered-data-length** to find how much buffered data there is in **async-io-state**.

See also **async-io-state-buffered-data-length**

“The Async-I/O-State API” on page 412

---

**async-io-state-handshake**

**Function**

**Summary**
Perform a SSL handshake on an **async-io-state**.

**Package**
**comm**

**Signature**
```lisp
async-io-state-handshake async-io-state callback &optional timeout
```

**Arguments**
- **async-io-state** A **socket-stream**.
- **callback** A function designator for a function with two arguments.
- **timeout** nil or a real.

**Values**
None.

**Description**
The function **async-io-state-handshake** performs a handshake on **async-io-state**, which must be attached to SSL.

When the handshake has finished successfully or failed, **callback** is called with two arguments: **async-io-state** and an error-indicator. The error-indicator is nil when successful, otherwise it is a list suitable for an error call as in `(apply 'error error-indicator)`. The **async-io-state-read-status** of **async-io-state** is also set appropriately.

If the handshake does not finish in **timeout** seconds, **callback** is called with non-nil error-indicator: (**Handshake timed out**), and the **async-io-state-read-status** is set to :timeout. If the other side closes the socket cleanly, **callback** is called error-indicator: (**SSL connection closed**),
and the async-io-state-read-status is set to :eof. Other cases indicate an actual error in the handshake.

async-io-state-handshake must not be called when there is any other operation on async-io-state and new operations on async-io-state must not be started until callback has been called.

Notes
If SSL was attached with ssl-side :both, then you will need to specify which side to take in the handshake by calling ssl-set-accept-state or ssl-set-connect-state with the ssl-pointer return by socket-stream-ssl.

See also
async-io-state
“Using SSL” on page 418

async-io-state-max-read

Summary
Accesses the maximum bytes to read of an async-io-state.

Package
comm

Signature
async-io-state-max-read async-io-state => max-read

setf (async-io-state-max-read async-io-state) max-read => max-read

Arguments
max-read An integer.

Values
max-read An integer.

Description
The accessor async-io-state-max-read is used to read and write the maximum bytes to try to read of async-io-state. max-read is an integer specifying the maximum number of bytes to try to read between calls to the callback in async-io-state-read-with-checking.
See also async-io-state
async-io-state-read-with-checking
“The Async-I/O-State API” on page 412

async-io-state-name

Accessor

Summary
Accesses the name of an async-io-state.

Package
comm

Signature
async-io-state-name async-io-state => name
(setf (async-io-state-name async-io-state) name => name)

Arguments
name A Lisp object.

Values
name A Lisp object.

Description
The accessor async-io-state-name is used to access the name of async-io-state. This names the state for debugging purposes.

See also async-io-state
create-async-io-state
“The Async-I/O-State API” on page 412

async-io-state-object

Function

Summary
Returns the object used to create an async-io-state.

Package
comm

Signature
async-io-state-object async-io-state => object
Arguments


Values

object  A socket-stream or an integer.

Description

The function async-io-state-object returns the object that was supplied in the call to create-async-io-state that created async-io-state.

See also
async-io-state
create-async-io-state
“The Async-I/O-State API” on page 412

async-io-state-old-length  Function

Summary

Returns the old length of an async-io-state.

Package

comm

Signature

async-io-state-old-length async-io-state => old-length

Arguments


Values

old-length  An integer.

Description

The function async-io-state-old-length is used to get the old length of async-io-state. old-length is an integer specifying the length of the old part in the buffer, that is the part that was seen in the previous invocation of the callback in async-io-state-read-with-checking.

See also
async-io-state
async-io-state-read-with-checking
“The Async-I/O-State API” on page 412
async-io-state-peer-address  

**Function**

**Summary**  
Returns the local address and port number for an `async-io-state` state that has a connected socket.

**Package**  
`comm`

**Signature**  
`async-io-state-peer-address  async-io-state => address, port`

**Arguments**  
- `async-io-state`  
  An `async-io-state`.

**Values**  
- `address`  
  An integer, an `ipv6-address` or `nil`.  
- `port`  
  An integer or `nil`.

**Description**  
The function `async-io-state-peer-address` returns the remote address and port number for `async-io-state` if it has a socket which is connected.

- `address` is the remote host address of the socket in the state.
- `port` is the remote port number of the socket in the state.

For an unconnected socket both `address` and port are `nil`.

**See also**  
- `async-io-state-address`  
- `get-socket-peer-address`  
- “The Async-I/O-State API” on page 412

---

async-io-state-read-buffer  

**Function**

**Summary**  
Asynchronously fills a buffer with bytes read from an `async-io-state`.

**Package**  
`comm`

**Signature**  
`async-io-state-read-buffer  async-io-state buffer callback &key start end timeout error-callback user-info`
Arguments

- **async-io-state**: An *async-io-state*.
- **buffer**: A **cl:base-string** or an 8-bit **cl:simple-array**.
- **callback**: A function designator for a function of 3 arguments.
- **start**: A lower bounding index designator for **buffer**.
- **end**: An upper bounding index designator for **buffer**.
- **timeout**: **nil** or a positive real.
- **error-callback**: A function designator for a function of 3 arguments, or **nil**.
- **user-info**: A Lisp object.

Values

None.

Description

The function **async-io-state-read-buffer** asynchronously fills the buffer **buffer** between **start** and **end** with bytes read from **async-io-state**. When buffer is full (between **start** and **end**) or the **async-io-state-read-timeout** of **async-io-state** has passed, the **callback** is called like this:

```
callback async-io-state buffer number-of-bytes-read
```

If an error occurs during the I/O operation and **error-callback** is non-nil, then **error-callback** is called with these same arguments:

```
error-callback async-io-state buffer number-of-bytes-read
```

If **error-callback** is **nil**, then **callback** is called, so it should check for errors using **async-io-state-read-status**.

The default value of **start** is 0. The default value of **end** is the length of **buffer**.
If the operation does not finish within the state's `async-io-state-read-timeout` period then state's `async-io-state-read-status` is set to `:timeout` and `callback` is called.

If `timeout` or `user-info` are supplied then they set `async-io-state-read-timeout` and `async-io-state-user-info` in `async-io-state` for this and subsequent operations.

If another read operation on the state is in progress, an error is signaled.

See also
- `async-io-state-write-buffer`
- `async-io-state-read-with-checking`
- “The Async-I/O-State API” on page 412

**async-io-state-read-status**
**async-io-state-write-status**

### Functions

**Summary**
Returns the read or write status of an `async-io-state`.

**Package**
`comm`

**Signature**
- `async-io-state-read-status` `async-io-state` => `read-status`
- `async-io-state-write-status` `async-io-state` => `write-status`

**Arguments**

**Values**
- `read-status`  `nil`, `:eof`, `:timeout` or an error value.
- `write-status`  `nil`, `:eof`, `:timeout` or an error value.

**Description**
The function `async-io-state-read-status` returns the read status of `async-io-state`. `read-status` is `nil` for a working socket, `:eof` for end of file, `:timeout` if a timeout has occurred or some other values meaning an error has occurred.
The function `async-io-state-write-status` returns the write status of `async-io-state`. `write-status` is `nil` for a working socket, `:eof` for end of file, `:timeout` if a timeout has occurred or some other values meaning an error has occurred.

See also `async-io-state`  
"The Async-1/O-State API" on page 412

**async-io-state-read-timeout**  
**async-io-state-write-timeout**  
*Accessors*

**Summary**  
Accesses the read and write timeouts of an `async-io-state`.

**Package**  
`com`

**Signature**  
`async-io-state-read-timeout async-io-state => read-timeout`

`setf (async-io-state-read-timeout async-io-state) read-timeout`

`async-io-state-write-timeout async-io-state => write-timeout`

`setf (async-io-state-write-timeout async-io-state) write-timeout`

**Arguments**  
`async-io-state`  
*An `async-io-state`.*

`read-timeout`  
`nil` or a positive real.

`write-timeout`  
`nil` or a positive real.

**Values**  
`read-timeout`  
`nil` or a positive real.

`write-timeout`  
`nil` or a positive real.

**Description**  
The accessor `async-io-state-read-timeout` is used to read and write the read timeout of `async-io-state`. `read-timeout` is `nil` if there is no timeout and otherwise is a timeout in seconds.
The accessor `async-io-state-write-timeout` is used to read and write the write timeout of `async-io-state`. `write-timeout` is `nil` if there is no timeout and otherwise is a timeout in seconds.

See also

- `async-io-state`
- `create-async-io-state`
- “The Async-I/O-State API” on page 412

### `async-io-state-read-with-checking` Function

**Summary**
Repeatedly tries to read bytes from an `async-io-state`, and invokes a callback.

**Package**
`comm`

**Signature**
```
async-io-state-read-with-checking async-io-state callback &key timeout max-read error-callback user-info element-type
```

**Arguments**
- `callback` A function designator for a function of 3 arguments, or `nil`.
- `timeout` `nil` or a positive real.
- `max-read` A positive integer.
- `error-callback` A function designator for a function of 3 arguments, or `nil`.
- `user-info` A Lisp object.
- `element-type` A type specifier.

**Values**
None.

**Description**
The function `async-io-state-read-with-checking` repeatedly tries to read up to `async-io-state-max-read`
more bytes from async-io-state, append them to the internal buffer and call callback like this:

callback async-io-state buffer end

async-io-state is the argument to async-io-state-read-with-checking, buffer is a cl:simple-array of element type element-type containing data from index 0 up to end, and end is a positive integer indicating the end of the filled part of buffer.

The buffer must not be modified or accessed outside the scope of the callback or after async-io-state-discard or async-io-state-finish have been called.

The element type of buffer is element-type, which can be base-char, (unsigned-byte 8) or (signed-byte 8). The default value of element-type is base-char.

The callback is responsible for processing the data in the buffer and optionally indicating that the read operation is complete as follows:

- The function async-io-state-old-length can be used to find the length of the old part in the buffer, that is the part that contained data in the previous call to the callback. When async-io-state-read-with-checking is called, it resets the old length to 0, so async-io-state-old-length returns 0 in the first invocation of callback.

- You can use async-io-state-discard with discard between 0 and end to discard the first discard bytes of the buffer. This is typically done when the callback has processed some of the bytes and does not want to see them again. Until bytes are discarded explicitly, they are accumulated in the buffer for subsequent calls to the callback.

- When the callback decides that the operation is complete, it needs to call async-io-state-finish. This optionally discards bytes as described above, and keeps the remaining bytes for future read operations from async-io-state.
If the operation does not finish within the state's `async-io-state-read-timeout` period then the callback is called with the state's `async-io-state-read-status` set to `:timeout`.

If an error occurs during the I/O operation and `error-callback` is non-nil, then `error-callback` is called like this:

```
error-callback async-io-state buffer end
```

If `error-callback` is `nil`, then `callback` is called, so it should check for errors using `async-io-state-read-status`.

If `timeout`, `max-read` or `user-info` are supplied then they set `async-io-state-read-timeout`, `async-io-state-max-read` and `async-io-state-user-info` in `async-io-state` for this and subsequent operations.

If another read operation is in progress on the state, an error is signaled.

**Notes**

Once the callback has called `async-io-state-finish` it can start further reading operations on `async-io-state`. The accessors `async-io-state-read-timeout`, `async-io-state-max-read` and `async-io-state-user-info` can be used to read and write the corresponding values in the callback.

**Examples**

Reading http headers, which are separated from the http body by two consecutive newlines. We assume these functions:

1. `my-parse-http-headers` which takes a buffer, start and end and returns a parsed headers-object.
2. `my-read-http-body` takes an `async-io-state`, headers-object and a user-defined object and reads the body via the Async-IO-State API.
3. `my-record-socket-error` which takes a user defined object and the error flag and handles a socket error.
4. `find-nn-in-buffer` which takes buffer, start and end and returns the index of the first two consecutive newlines if any.
The callback is defined like this:

```lisp
(defun http-header-reading-callback (state buffer end)
  (if-let (cannot-read
    (async-io-state-read-status state))
    (my-record-socket-error
      (async-io-state-user-info state)
     cannot-read)
   (let ((start (async-io-state-old-length state)))
    (let ((start-search-for-nn
      (if (zerop start) 0 (1- start)))
     (when-let (h-end (find-nn-in-buffer
       buffer
       start-search-for-nn
       end))
    (let ((h-object (my-parse-http-headers
      buffer 0 h-end)))
     (async-io-state-finish state (+ h-end 2))
     (my-read-http-body
       state
       h-object
       (async-io-state-user-info state)))))))
```

The callback is used like this:

```lisp
(async-io-state-read-with-checking
 state
 'http-header-reading-callback)
```

See also

- `async-io-state-read-buffer`
- `async-io-state-write-buffer`
- “The Async-I/O-State API” on page 412

---

**async-io-state-receive-message**

*Function*

**Summary**: Asynchronously receives a message from a socket.

**Package**: `comm`

**Signature**:

```
async-io-state-receive-message async-io-state buffer callback
&key start end timeout error-callback needs-address user-info
```
## Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>buffer</code></td>
<td>A <code>cl:base-string</code> or an 8-bit <code>cl:simple-array</code>.</td>
</tr>
<tr>
<td><code>callback</code></td>
<td>A function designator.</td>
</tr>
<tr>
<td><code>start</code></td>
<td>A lower bounding index designator for <code>buffer</code>.</td>
</tr>
<tr>
<td><code>end</code></td>
<td>An upper bounding index designator for <code>buffer</code>.</td>
</tr>
<tr>
<td><code>timeout</code></td>
<td><code>nil</code> or a non-negative real.</td>
</tr>
<tr>
<td><code>error-callback</code></td>
<td>A function designator.</td>
</tr>
<tr>
<td><code>needs-address</code></td>
<td>A boolean.</td>
</tr>
<tr>
<td><code>user-info</code></td>
<td>A Lisp object.</td>
</tr>
</tbody>
</table>

## Values

None.

## Description

The function `async-io-state-receive-message` starts a read operation of "receiving" on `async-io-state`, which means that when there is input on the socket it calls `recv` or `recvfrom` to read the data into `buffer` between `start` and `end`.

The default value of `start` is 0. The default value of `end` is the length of `buffer`.

`callback` should be a function of 3 or 5 arguments. If the reading succeeds and `needs-address` is `nil`, then `callback` is called with this signature:

```
callback async-io-state buffer number-of-bytes-read
```

If the reading succeeds and `needs-address` is non-nil, then `callback` is called with this signature:

```
callback async-io-state buffer number-of-bytes-read ip-address port-number
```

where `ip-address` and `port-number` are the socket address of the sender, and can be used as the `hostspec` and `service` when...
required. Typically these are used in `async-io-state-send-message-to-address` to send a message back to the sender.

The default value of `needs-address` is `nil`.

`error-callback`, `timeout`, `start`, `end` and `user-info` have the same meaning as in `async-io-state-read-buffer`.

Notes

2. The socket may or may not be connected.

Example

(example-edit-file "async-io/udp")

See also
`create-async-io-state-and-udp-socket`
`create-async-io-state-and-connected-udp-socket`
`async-io-state-send-message`
`async-io-state-send-message-to-address`
“The Async-I/O-State API” on page 412

async-io-state-send-message

Function

Summary
Asynchronously sends a message on a connected socket.

Package
`comm`

Signature
```
async-io-state-send-message async-io-state buffer callback
&key start end timeout error-callback user-info
```

Arguments
- `buffer` A `cl:base-string` or an 8-bit `cl:simple-array`. 
**Values**
None.

**Description**
The function `async-io-state-send-message` asynchronously sends a message from `buffer` between `start` and `end`. The socket in `async-io-state` must be connected. When the send is successful, `callback` is called with `async-io-state` as its only argument.

The default value of `start` is 0. The default value of `end` is the length of `buffer`.

`error-callback`, `timeout`, `start`, `end` and `user-info` have the same meaning as in `async-io-state-write-buffer`.

**Notes**

The contents of `buffer` must not be changed until `callback` has been called.

For unconnected UDP sockets, use `async-io-state-send-message-to-address`.

---

**callback**
A function designator for a function of 1 argument.

**start**
A lower bounding index designator for `buffer`.

**end**
An upper bounding index designator for `buffer`.

**timeout**
`nil` or a non-negative real.

**error-callback**
A function designator.

**user-info**
A Lisp object.
async-io-state-send-message-to-address

Function

Summary
Asynchronously sends a message on an unconnected socket.

Package
comm

Signature
async-io-state-send-message-to-address (async-io-state hostspec service buffer callback &key start end timeout error-callback user-info)

Arguments


hostspec, service Specify the socket address to send to in the standard way.

buffer A cl:base-string or an 8-bit cl:simple-array.

callback A function designator for a function of 1 argument.

start A lower bounding index designator for buffer.

derg An upper bounding index designator for buffer.

timeout nil or a non-negative real.

error-callback A function designator.

user-info A Lisp object.

Values None.
The function `async-io-state-send-message-to-address` asynchronously sends a message from `buffer` between `start` and `end` to the socket address which is specified by `hostspec` and `service`. For the interpretation of `hostspec` and `service` see “Specifying the target for connecting and binding a socket” on page 408.

The default value of `start` is 0. The default value of `end` is the length of `buffer`.

The socket in `async-io-state` must not be connected. When the sending is successful, `callback` is called with `async-io-state` as its only argument.

`error-callback`, `timeout`, `start`, `end` and `user-info` have the same meaning as in `async-io-state-write-buffer`.


2. The contents of `buffer` must not be changed until `callback` has been called.

3. If `hostspec` is a host name (that is a string not specifying an IP address), then `async-io-state-send-message-to-address` uses the family of the socket to decide whether to look for IPv6 or IPv4 addresses. If `async-io-state` was created by `create-async-io-state`, the `ipv6` argument to `create-async-io-state` must match the family of the socket for `async-io-state-send-message-to-address` to work.


(Example: `example-edit-file "async-io/udp"`)
async-io-state-ssl  

*Function*

**Summary**
Accesses the SSL attached to an async-io-state.

**Package**
comm

**Signature**
async-io-state-ssl async-io-state => ssl-pointer

**Arguments**
async-io-state  
An async-io-state.

**Values**
ssl-pointer  
A foreign pointer of type ssl-pointer, or nil.

**Description**
The function async-io-state-ssl accesses the SSL that is attached to the async-io-state async-io-state. It returns nil if SSL is not attached.

See also
async-io-state
ssl-pointer

async-io-state-ssl-side  

*Function*

**Summary**
Accesses the ssl-side of an async-io-state.

**Package**
comm

**Signature**
async-io-state-ssl-side async-io-state => ssl-side

**Arguments**
async-io-state  
An async-io-state.
The function \texttt{async-io-state-ssl-side} accesses the ssl-side of the \texttt{async-io-state async-io-state}.

It returns \texttt{nil} if SSL is not attached.

\textbf{Note} \texttt{async-io-state-ssl-side} is useful as a predicate for testing if an \texttt{async-io-state} has SSL attached.

\textbf{See also} \texttt{async-io-state}

\textbf{async-io-state-write-buffer} \hspace{1em} \textit{Function}

\textbf{Summary} \hspace{1em} Asynchronously writes a buffer to an \texttt{async-io-state}

\textbf{Package} \hspace{1em} \texttt{comm}

\textbf{Signature} \hspace{1em} \texttt{async-io-state-write-buffer async-io-state buffer callback &key start end timeout error-callback user-info}

\textbf{Arguments} \hspace{1em} \texttt{async-io-state} An \texttt{async-io-state}.

\textit{buffer} \hspace{1em} A \texttt{cl:base-string} or an 8-bit \texttt{cl:simple-array}.

\textit{callback} \hspace{1em} A function designator for a function of 3 arguments.

\textit{start} \hspace{1em} A lower bounding index designator for \textit{buffer}.

\textit{end} \hspace{1em} An upper bounding index designator for \textit{buffer}.

\textit{timeout} \hspace{1em} \texttt{nil} or a positive real.

\textit{error-callback} \hspace{1em} A function designator for a function of 3 arguments, or \texttt{nil}.

\textit{user-info} \hspace{1em} A Lisp object.
Values
None.

Description
The function \texttt{async-io-state-write-buffer} asynchronously writes the part of buffer \texttt{buffer} between indexes \texttt{start} and \texttt{end} to \texttt{async-io-state}. When this writing has succeeded or the state's \texttt{async-io-state-write-timeout} has passed, \texttt{callback} is called like this:

\texttt{callback async-io-state buffer number-of-bytes-written}

The default value of \texttt{start} is 0. The default value of \texttt{end} is the length of \texttt{buffer}.

If an error occurs during the I/O operation and \texttt{error-callback} is non-nil, then \texttt{error-callback} is called with these same arguments:

\texttt{error-callback async-io-state buffer number-of-bytes-written}

If \texttt{error-callback} is \texttt{nil}, then \texttt{callback} is called, so it should check for errors using \texttt{async-io-state-write-status}.

If the operation does not finish within the state's \texttt{async-io-state-write-timeout} period then the state's \texttt{async-io-state-write-status} is set to \texttt{:timeout} and \texttt{callback} is called.

If \texttt{timeout} or \texttt{user-info} are supplied then they set the state's \texttt{async-io-state-write-timeout} and \texttt{async-io-state-user-info} for this and subsequent operations.

See also
\texttt{async-io-state-read-buffer}
\texttt{async-io-state-read-with-checking}
“The Async-I/O-State API” on page 412

\texttt{async-io-state-user-info}

\texttt{Accessor}

Summary
Accesses the user info of an \texttt{async-io-state}.

Package
\texttt{comm}
The accessor `async-io-state-user-info` is used to read and write the `user-info` of `async-io-state`. LispWorks itself does not use this value for any purpose.

See also
- `async-io-state`
- `create-async-io-state`
- “The Async-I/O-State API” on page 412

### attach-ssl

**Function**

Attaches SSL to a socket stream.

**Summary**

Attaches SSL to a socket stream.

**Package**

`comm`

**Signature**

`attach-ssl socket-stream &key ssl-ctx ssl-side ctx-configure-callback ssl-configure-callback handshake-timeout tlsext-host-name => ssl`

**Arguments**

- `socket-stream` A socket-stream.
- `ssl-ctx` A symbol or a foreign pointer.
- `ssl-side` One of the keywords `:client`, `:server` or `:both`. 
**ctx-configure-callback**

A function designator or nil. The default value is nil.

**ssl-configure-callback**

A function designator or nil. The default value is nil.

**handshake-timeout**

A real or nil (the default).

**tlsext-host-name**

A string or nil.

**Values**

- **ssl**: A foreign pointer of type ssl-pointer.

**Description**

The function **attach-ssl** attaches SSL to the socket-stream.

The allowed values and meaning of the keyword arguments are as described for **socket-stream**.

Note that **attach-ssl** is used by

(make-instance 'comm:socket-stream :ssl-ctx ...)

and by

(comm:open-tcp-stream ... :ssl-ctx ...)

but you can also call it explicitly.

Before starting to create objects, **attach-ssl** ensures the SSL library (by calling **ensure-ssl**) and calls **do-rand-seed** to seed the Pseudo Random Number Generator (PRNG), so normally you do not need to worry about these.

**ssl-side**, **ssl-ctx**, **ctx-configure-callback**, **ssl-configure-callback** and **handshake-timeout** are interpreted as described in “Keyword arguments for use with SSL” on page 421. After this, **SSL_set_fd** is used to attach the SSL to the socket and this is recorded in the socket stream.

The default value of **ssl-ctx** is t and the default value of **ssl-side** is :server.
If `tlsext-host-name` is non-nil, then the SNI extension in the SSL connection is set to its value.

When a `socket-stream` is closed, `detach-ssl` is called with `:retry-count nil`, which, if the stream is attached to SSL, calls `SSL_shutdown` and then frees the object (or objects) that were automatically allocated.

If SSL is already attached to `socket-stream` then `attach-ssl` signals an error.

See also `detach-ssl`

### call-wait-state-collection

**Function**

**Summary**

Calls the functions associated with the active states in a `wait-state-collection`.

**Package**

`comm`

**Signature**

`call-wait-state-collection collection`

**Arguments**

`collection` A `wait-state-collection`.

**Description**

The function `call-wait-state-collection` calls the functions associated with the active states in `collection`, and perform any actions requested by messages that arrive from other processes.

**Notes**

Typically you would not call `call-wait-state-collection` yourself, but it will be called by `loop-processing-wait-state-collection`.

**See also**

`create-and-run-wait-state-collection`

`loop-processing-wait-state-collection`

“The Async-I/O-State API” on page 412
close-accepting-handle

Function

Summary
Closes an accepting handle.

Package
comm

Signature
close-accepting-handle accepting-handle &optional callback

Arguments
accepting-handle An accepting-handle.
callback A function designator or nil.

Description
The function close-accepting-handle closes the accepting handle accepting-handle. In particular, it closes the socket which frees up the port that the socket is bound to.

accepting-handle has to be an accepting handle, currently that means the result of accept-tcp-connections-creating-async-io-states.

If callback is non-nil, it must be a function of one argument. callback is called after closing the handle, with the collection which was supplied to accept-tcp-connections-creating-async-io-states which created the handle.

close-accepting-handle is asynchronous. To do something which is guaranteed to happen after the socket is closed, use callback.

Notes
callback is called on the collection process, so it should not do much work.

See also
accepting-handle
accept-tcp-connections-creating-async-io-states
close-async-io-state  

**Function**

**Summary**  Closes an `async-io-state` and removes it from any internal structures.

**Package**  comm

**Signature**  

`close-async-io-state async-io-state &key keep-alive-p => buffered-data-length`

**Arguments**  

- `keep-alive-p`  A generalized boolean.

**Values**  

- `buffered-data-length`  A non-negative integer.

**Description**  The function `close-async-io-state` closes `async-io-state` and removes it from any internal structures. Once `async-io-state` has been closed, you cannot perform I/O operations on it.

By default, `close-async-io-state` also closes the object in `async-io-state` (that is, the argument to `create-async-io-state`). This closing can be prevented by supplying true for `keep-alive-p`, so you can perform further I/O operations on that object. In this case you will need to close object later.

`async-io-state` may contain some buffered data that it read from the object but did not use yet. The return value is the length of such data and you can use `async-io-state-get-buffered-data` to get it.

**Notes**  If `async-io-state` is attached to SSL, then it is detached. This occurs even if `keep-alive-p` is true.

**See also**  “The Async-I/O-State API” on page 412
close-wait-state-collection

Summary
Closes a wait-state-collection and all of its states.

Package
comm

Signature
close-wait-state-collection collection

Arguments
collection A wait-state-collection.

Description
The function close-wait-state-collection closes all of the states in collection. That means that the underlying communication object is closed and the async-io-state objects that are currently associated with collection cannot be used for further I/O and will not receive callbacks anymore.

Notes
close-wait-state-collection does not do anything that affects further processing in collection. In particular, you can add new states to the collection afterwards, and waiting and calling, either by loop-processing-wait-state-collection or wait-for-wait-state-collection and call-wait-state-collection, can continue. loop-processing-wait-state-collection does not stop if close-wait-state-collection is called inside it.

close-wait-state-collection is a “nasty” call, because it just kills any async-io-state associated with the collection. Normally should only be used only when you stop using the collection.

close-wait-state-collection cannot be called on a collection in parallel to itself or loop-processing-wait-state-collection, wait-for-wait-state-collection or call-wait-state-collection. It can be called inside the scope of call-wait-state-collection, and loop-processing-wait-state-collection.
You can use `apply-in-wait-state-collection-process` to cause execution inside the scope of `call-wait-state-collection`.

See also `create-and-run-wait-state-collection` "The Async-I/O-State API" on page 412

**create-and-run-wait-state-collection**

*Function*

**Summary** Creates and runs a `wait-state-collection`.

**Package** `comm`

**Signature**

```lisp
create-and-run-wait-state-collection name &key handler
with-backtrace => wait-state-collection
```

**Arguments**

- `name` A Lisp object that names the collection. It is used only for printing.
- `handler` `nil, t, the keyword :abort` or a function.
- `with-backtrace` The keyword:`bug-form, t, the keyword :quick, or nil.`

**Values**


**Description** The function `create-and-run-wait-state-collection` creates and runs a `wait-state-collection`.

`create-and-run-wait-state-collection` creates a `wait-state-collection` and then starts a new process which calls `loop-processing-wait-state-collection` on the new `wait-state-collection` (and therefore activates it), and returns it as `wait-state-collection`. The new process has process name "Loop Collection name". When `loop-processing-wait-state-collection` exits, `wait-state-collection` is closed and the other process exits too.
You can use `wait-state-collection-stop-loop` to make `loop-processing-wait-state-collection` exit, and hence close `wait-state-collection` and make the process go away. Calling `process-terminate` on the process itself can also be used, because it will use `wait-state-collection-stop-loop`.

`handler` specifies handling of errors that occur on the process in which the collection is run. The values have the following effects:

- **nil**: No handling.
- **:abort**: Abort (calls the function `cl:abort`).
- **t**: Print the condition to the standard output, and unless `with-backtrace` is `nil` produces a backtrace, and then aborts.

A function must be a function of three arguments when `with-backtrace` is non-nil, or two arguments when `with-backtrace` is `nil`. When a serious condition is signaled, the handler is called inside the context of the error (like a handler in `cl:handler-bind`).

When `with-backtrace` is non-nil:

```
handler object condition backtrace-string
```

When `with-backtrace` is `nil`:

```
handler object condition
```

The `object` argument is the object that is responsible for the error. Currently this is always the `async-io-state` with which the callback that caused the error is associated. If there is an error outside a callback (which should not happen, unless there is a bug), then `object` is `nil`. `condition` is the condition that is signaled. `backtrace-string` is a string which is the result of producing a backtrace. If the handler returns, `(cl:abort)` is called.
with-backtrace controls whether a backtrace is produced when handler is t or a function. It is passed to output-backtrace as the first argument. See output-backtrace for details.

The default value of handler is nil. The default value of with-backtrace is :bug-form.

wait-state-collection can be used immediately by passing it to one of the create-async-io-state* functions.

Notes


2. Aborting by the handler is done by calling (cl:abort), which aborts to the closest enclosing abort restart. If your code establishes such a restart around the error, the aborting will abort to it. Otherwise it will abort back to the loop of waiting and calling.

3. Real applications will probably always pass the handler.

4. While the handler is run, no further processing is done in the collection. Therefore the handler should not do a significant amount of work.

See also

wait-state-collection-stop-loop
create-async-io-state-and-connected-tcp-socket
create-async-io-state-and-connected-udp-socket
create-async-io-state
create-async-io-state-and-udp-socket
accept-tcp-connections-creating-async-io-states

create-async-io-state

Function

Summary

Creates an async-io-state for a socket.

Package

comm
create-async-io-state collection object &key read-timeout write-timeout user-info udp ipv6 name queue-output => async-io-state

Arguments

- **collection**: A wait-state-collection.
- **object**: A socket-stream or an integer.
- **read-timeout**: nil or a positive real.
- **write-timeout**: nil or a positive real.
- **user-info**: A Lisp object.
- **udp**: nil, t, or the keyword :connected.
- **ipv6**: A boolean.
- **name**: A Lisp object.
- **queue-output**: A boolean.

Values

- **async-io-state**: An async-io-state.

Description

The function `create-async-io-state` creates an `async-io-state` for the object `object`. If `object` is an integer, then it is assumed to be a socket handle (a file descriptor on Unix-like systems). If `object` is a socket-stream, then the `async-io-state` contains its socket.


If `udp` is non-nil and `object` is a socket, then this tells `create-async-io-state` that the socket is a UDP socket (rather than TCP). If `udp` is :connected, this also tells `create-async-io-state` that the socket is a connected socket, which affects whether you can use `async-io-state-send-message` (connected) or `async-io-state-send-message-to-address` (unconnected). When `object` is a stream, it is always assumed
to be a TCP socket, regardless of the value of `udp`. The default value of `udp` is `nil`.

`ipv6` tells `create-async-io-state` whether the socket was made as an IPv6 socket (with `AF_INET6`) or IPv4 (with `AF_INET`). This makes a difference only for unconnected UDP sockets (it tells `async-io-state-send-message-to-address` when called with a host name whether to look for IPv6 or IPv4 addresses).

`queue-output` controls what happens if you try to perform a write operation on the state while another write operation is ongoing. When `queue-output` is `nil` this will cause an error. When `queue-output` is non-nil, the second write operation is queued and actually executed later. The default value of `queue-output` is `nil`.

After calling `create-async-io-state`, `object` should not be used directly for I/O in the same direction (read or write) until `close-async-io-state` has been called.

See also

- `create-async-io-state-and-connected-tcp-socket`
- `accept-tcp-connections-creating-async-io-states`
- “The Async-I/O-State API” on page 412

### create-async-io-state-and-connected-tcp-socket

**Function**

**Summary**

Creates an `async-io-state` which attempts to make a TCP connection.

**Package**

`comm`

**Signature**

```go
create-async-io-state-and-connected-tcp-socket collection host service callback &key read-timeout write-timeout user-info connect-timeout local-address local-port keepalive nodelay name queue-output => async-io-state ssl-ctx ctx-configure-callback ssl-configure-callback handshake-timeout
```

**Arguments**

- `collection` A `wait-state-collection`. 
**host**
An integer or a string or an `ipv6-address` object.

**service**
A string or a fixnum.

**callback**
A function designator for a function of two arguments.

**read-timeout**
nil or a positive real.

**write-timeout**
nil or a positive real.

**user-info**
A Lisp object.

**connect-timeout**
nil or a positive real.

**local-address**
nil, an integer, a string or an `ipv6-address` object.

**local-port**
nil, a string or a fixnum.

**keepalive**
A generalized boolean.

**nodelay**
A generalized boolean.

**queue-output**
A boolean.

**name**
A Lisp object.

**ssl-ctx**
A symbol or a foreign pointer.

**ctx-configure-callback**
A function designator or nil. The default value is nil.

**ssl-configure-callback**
A function designator or nil. The default value is nil.

**handshake-timeout**
A real or nil (the default).

**Values**


565
The function `create-async-io-state-and-connected-tcp-socket` creates an `async-io-state` which attempts to make a TCP connection to `host` on port `service` within `connect-timeout` seconds. `host` and `service` are interpreted as described in “Specifying the target for connecting and binding a socket” on page 408.

When the connection has been made, `callback` is called with arguments `async-io-state` and `nil`. Normally `callback` will start asynchronous I/O by calling `async-io-state-read-buffer`, `async-io-state-write-buffer` or `async-io-state-read-with-checking`.

If no connection can be made, `callback` is called with `async-io-state` and a list of a format control-string and args, suitable for applying to `format`. In general, it doesn’t make much sense for the callback to call `error` within `callback`, so it should report the problem in some way, typically by writing to some log. It may also need to inform the user interactively, but that needs be done in another process, and is better done by using some kind of an end-user dialog rather than invoking `error`.

`local-address` and `local-port` are used to bind the local side of the socket to a particular address and/or port if non-nil.

`keepalive` and `nodelay` set the SO_KEEPALIVE and TCP_NODELAY in the socket. The default value of `keepalive` is `nil`. The default value of `nodelay` is `t`.

`queue-output` controls what happens if you try to perform a write operation on the state while another write operation is ongoing. When `nil`, this will cause an error. When non-nil, the second write operation is queued and actually executed later. The default value of `queue-output` is `nil`.

The default value of name is a string "Connected TCP”.

ssl-ctx, ctx-callback-ssl, ctx-callback-ssl and hand-shake-timeout are interpreted as described in “Keyword arguments for use with SSL” on page 421. Unlike the other ways of creating a socket stream with SSL processing, create-async-io-state-and-connected-tcp-socket does not take the ssl-side argument and always uses the value :client.

Once the connection has been made, you can get the socket by calling async-io-state-object on async-io-state (see async-io-state)

See also
create-async-io-state
accept-tcp-connections-creating-async-io-states
“The Async-I/O-State API” on page 412

create-async-io-state-and-connected-udp-socket

Function

Summary
Creates an async-io-state where the object is a connected UDP socket.

Package
comm

Signature
create-async-io-state-and-connected-udp-socket collection
hostspec service &key queue-output name errorp ipv6 read-timeout write-timeout user-info local-address local-port => async-io-state

Arguments

collection A wait-state-collection to associate with the result.

hostspec, service Specify the socket address to connect to in the standard way.

ipv6 One of nil, t or the keyword :any.

errorp A boolean.

local-address, local-port
Specify the local socket address in the standard way.

- **queue-output**: A boolean.
- **read-timeout**: \texttt{nil} or a positive real.
- **write-timeout**: \texttt{nil} or a positive real.
- **user-info**: A Lisp object.
- **name**: A Lisp object.

**Values**

- **async-io-state**: An \texttt{async-io-state}.

**Description**

The function \texttt{create-async-io-state-and-connected-udp-socket} creates a new UDP socket, connects it to the socket address specified by \texttt{hostspec} and \texttt{service}, optionally binds it if \texttt{local-port} or \texttt{local-address} are non-nil, and then creates and returns an \texttt{async-io-state} object that can be used to perform I/O operations on the socket. The I/O operations are done using \texttt{async-io-state-receive-message} and \texttt{async-io-state-send-message}.

\texttt{hostspec} and \texttt{service} are interpreted as described in “Specifying the target for connecting and binding a socket” on page 408.

\texttt{local-address} and \texttt{local-port} are also interpreted as described in “Specifying the target for connecting and binding a socket” on page 408. Both values can be \texttt{nil}.

Connecting the socket affects the destination of messages sent using the \texttt{async-io-state}, and also restricts the origin of received messages.

When \texttt{ipv6} is :\texttt{any}, the system selects whether to use an IPv4 or IPv6 socket (normally it will be IPv4). When \texttt{ipv6} is \texttt{t} it forces using IPv6, and \texttt{nil} forces IPv4. The default value of \texttt{ipv6} is :\texttt{any}.

\texttt{queue-output} controls what happens if you try to perform a write operation on the state while another write operation is
ongoing. When queue-output is nil, this will cause an error. When queue-output is non-nil, the second write operation is queued and actually executed later. The default value of queue-output is t.


The default value of name is a string "Connected UDP".

When errorp is nil, create-async-io-state-and-connected-udp-socket returns nil for run time errors rather than signalling an error. The default value of errorp is t.

Notes

1. If you need an unconnected socket, use create-async-io-state-and-udp-socket.

2. The call to create-async-io-state-and-connected-udp-socket itself is synchronous.

3. You cannot use async-io-state-send-message-to-address with the result of create-async-io-state-and-connected-udp-socket (because the socket address to send to is already specified by connecting.)

See also async-io-state-receive-message
async-io-state-send-message
create-async-io-state-and-udp-socket
“The Async-I/O-State API” on page 412

create-async-io-state-and-udp-socket

Function

Summary Creates an async-io-state where object is an unconnected UDP socket.

Package comm
Signature  
create-async-io-state-and-udp-socket collection &key name errorp ipv6 queue-output read-timeout write-timeout user-info local-address local-port => result

Arguments  
collection  A wait-state-collection to associate with the returned async-io-state.
ipv6  One of nil, t, the keyword :any or the keyword :both.
errorp  A boolean.
local-address, local-port  Specify the local socket address in the standard way.
read-timeout  nil or a positive real.
write-timeout  nil or a positive real.
user-info  A Lisp object.
name  A Lisp object.
queue-output  A boolean.

Values  
result  An async-io-state or nil.

Description  
The function create-async-io-state-and-udp-socket creates an async-io-state where object is an unconnected UDP socket.
create-async-io-state-with-udp-socket creates a new UDP socket, optionally binds it if local-port or local-address is non-nil, and then creates and returns an async-io-state object that can be used to perform I/O operations on the socket. The I/O operations are performed using async-io-state-receive-message and async-io-state-send-message-to-address. local-address and local-port specify the local socket address as described in “Specifying the target for connecting and binding a socket” on page 408. Both values can be nil.
queue-output controls what happens if you try to perform a write operation on the state while another write operation is ongoing. When nil, this will cause an error. When non-nil, the second write operation is queued and actually executed later. The default value of queue-output is t.

When ipv6 is :any, the system selects whether to use an IPv4 or IPv6 socket (normally it will be IPv4). When ipv6 is t it forces using IPv6, and nil forces IPv4. The value :both means using IPv6, but also allow receiving messages in IPv4. The default value of ipv6 is :any.

When errorp is nil, create-async-io-state-and-udp-socket returns nil for run time errors rather than signalling an error. The default value of errorp is t.


The default value of name is a string "UDP".

Notes

1. If the socket is used to receive messages from unknown senders (that is as a server), then you need to bind the socket by supplying local-port. If the socket is only used to send messages then you do not need to bind it, because the recipient of the messages can find the socket's address if it needs to send a reply. You can supply local-address to restrict which connections are allowed.

2. You can find the source address of a message that is received using the result of create-async-io-state-and-udp-socket by supplying needs-address t to async-io-state-receive-message.

3. If you need to connect the socket, use create-async-io-state-and-connected-udp-socket instead.
4. The call to `create-async-io-state-and-udp-socket` itself is synchronous.

5. You cannot use `async-io-state-send-message` (without address) with the result of `create-async-io-state-and-udp-socket` (because the socket address to send to must be specified).

Example

(\example-edit-file \"async-io/udp\")

See also

`async-io-state-receive-message`

`async-io-state-send-message-to-address`

`create-async-io-state-and-connected-udp-socket`

“The Async-I/O-State API” on page 412

### connect-to-tcp-server

**Function**

**Summary**
Attempts to connect to a socket on a server.

**Package**
\comm

**Signature**

```
connect-to-tcp-server hostspec service &key errorp timeout
local-address local-port keepalive nodelay ipv6 => handle
```

**Arguments**

- `hostspec` An integer or a string or an `ipv6-address` object.
- `service` A string or a fixnum.
- `errorp` A boolean.
- `timeout` A positive number, or `nil`.
- `local-address` `nil`, an integer, a string or an `ipv6-address` object.
- `local-port` `nil`, a string or a fixnum.
- `keepalive` A generalized boolean.
- `nodelay` A generalized boolean.
The function `connect-to-tcp-server` attempts to connect to a socket on a server and returns a handle for the connection if successful. This handle can then be used as the socket when making a `socket-stream`.

The IP address to connect to is specified by `hostspec`, and the service to provide is specified by `service`. These two arguments are interpreted as described in “Specifying the target for connecting and binding a socket” on page 408.

If `errorp` is `nil`, failure to connect (possibly after `timeout` seconds) returns `nil`, otherwise an error is signaled.

`timeout` specifies a connection timeout. `connect-to-tcp-server` waits for at most `timeout` seconds for the TCP connection to be made. If `timeout` is `nil` it waits until the connection attempt succeeds or fails. On failure, `connect-to-tcp-server` signals an error or returns `nil` according to the value of `errorp`. To provide a timeout for reads after the connection is made, see `read-timeout` in `socket-stream`. The default value of `timeout` is `nil`.

If `local-address` is `nil` then the operating system chooses the local address of the socket. Otherwise the value is interpreted as for `hostspec` and specifies the local address of the socket. The default value of `local-address` is `nil`.

If `local-port` is `nil` then the operating system chooses the local port of the socket. Otherwise the string or fixnum value is interpreted as for `service` and specifies the local port of the socket. The default value of `local-port` is `nil`.

If `keepalive` is true, `SO_KEEPALIVE` is set on the socket. The default value of `keepalive` is `nil`.
If `nodelay` is true, **TCP_NODELAY** is set on the socket. The default value of `nodelay` is `t`.

`ipv6` specifies the address family to use when `hostspec` is a string. When `ipv6` is `:any`, **connect-to-tcp-server** uses either of IPv4 or IPv6. When `ipv6` is `t`, it uses only IPv6 addresses, and when `ipv6` is `nil` it tries only IPv4. The default value of `ipv6` is `:any`.

**Notes**

1. On Unix the name of the service can normally be found in `/etc/services`. If it is not there, the manual entry for services can be used to find it.

2. In most situations, **open-tcp-stream**, which returns a stream rather than a handle, is the more convenient interface. **connect-to-tcp-server** is useful when want to associate further information with the stream. You can define a subclass of **socket-stream**, connect using **connect-to-tcp-server**, and call **make-instance** with your subclass, passing the `handle` as the socket.

**See also**

socket-stream
open-tcp-stream

---

**destroy-ssl**

**Function**

**Summary**
Frees a SSL.

**Package**
`comm`

**Signature**
`destroy-ssl ssl-pointer`

**Arguments**

- **ssl-pointer**: A foreign pointer of type `ssl-pointer`.

**Description**
The function **destroy-ssl** frees the SSL pointed to by `ssl-pointer` and also frees any LispWorks cached values associated with it.
**destroy-ssl-ctx**

*Function*

**Summary**
Frees a SSL_CTX.

**Package**
comm

**Signature**
destroy-ssl-ctx ssl-ctx-pointer

**Arguments**
ssl-ctx-pointer  A foreign pointer of type ssl-ctx-pointer.

**Description**
The function destroy-ssl-ctx frees the SSL_CTX pointed to by ssl-ctx-pointer and also frees any LispWorks cached values associated with it.

**See also**
ssl-ctx-pointer

---

**detach-ssl**

*Function*

**Summary**
Detaches the SSL from a socket stream.

**Package**
comm

**Signature**
detach-ssl socket-stream &key retry-count retry-timeout

**Arguments**
socket-stream  A socket-stream.
retry-count  A non-negative integer.
retry-timeout  A non-negative real.

**Description**
The function detach-ssl detaches the SSL from the socket-stream socket-stream. If socket-stream is not attached to an SSL, detach-ssl just returns immediately. Otherwise, it detaches
the SSL from socket-stream, tries to shut down the SSL cleanly, and then frees the objects that were allocated by attach-ssl.

retry-count specifies how many additional times to call SSL_shutdown after the second to attempt to get a successful shutdown. The default value of retry-count is 5.

retry-timeout specifies the time in seconds to wait between each of the calls to SSL_shutdown. If it fails to get a successful shutdown after these attempts, detach-ssl signals an error. The default value of retry-timeout is 0.1.

Note that the shutdown calls happen after the SSL has been detached from socket-stream as far as LispWorks is concerned, so if an error occurs at this point and is aborted, socket-stream can be used in attach-ssl again (assuming that the peer can cope with this situation).

If retry-count is nil, detach-ssl does not try to get a successful shutdown call. This value is used when the stream is closed, but should not be used normally.

See also attach-ssl

do-rand-seed

Function

Summary Calls the SSL function RAND_seed.

Package comm

Signature do-rand-seed

Arguments do-rand-seed takes no arguments.

Values do-rand-seed returns no values.

Description The function do-rand-seed calls the SSL function RAND_seed with some suitable value, which is dependent in a
non-trivial way on the current time, the history of the current process and the history of the machine it is running on.

If the machine that it runs on has the file /dev/urandom, do-rand-seed does nothing.

See also attach-ssl

ensure-ssl

Summary Initializes SSL.

Package comm

Signature ensure-ssl &key library-path already-done

Arguments

library-path A string or a list of strings.

already-done A generalized boolean.

Description The function ensure-ssl initializes SSL. If it was already called in the image, ensure-ssl does nothing. Otherwise it loads the library, calls SSL_library_init, calls SSL_load_error_strings and performs internal initializations.

If already-done is true, ensure-ssl does only the internal initializations. The default value of already-done is nil.

If library-path is passed, it needs to be either a string, specifying one library, or a list of strings specifying multiple libraries. The default value of library-path is platform-specific. The initial default value is described in “How LispWorks locates the OpenSSL libraries” on page 420. This default may be changed by calling set-ssl-library-path.

See also openssl-version

set-ssl-library-path
get-default-local-ipv6-address

Summary  Gets the default IPv6 address of the local host.

Package  comm

Signature  get-default-local-ipv6-address => result

Arguments  None.

Values  result  An ipv6-address object or nil.

Description  The function get-default-local-ipv6-address tries to find the local default IPv6 address and if successful returns it.

See also  ipv6-address  get-ip-default-zone-id

get-host-entry

Summary  Returns address (IPv6 and IPv4) or name information about a given host.

Package  comm

Signature  get-host-entry host &key fields ipv6 v4mapped addrconfig numerichost avoid-reverse-lookup => field-values

Arguments  host  A number or a string.

fields  A list of keywords.

ipv6  nil, t or the keyword :any.

avoid-reverse-lookup  A generalized boolean.

v4mapped  A boolean.
The function `get-host-entry` returns address or name information about the given host. By default, it tries to find addresses both of IPv6 and IPv4. It uses whatever host naming services are configured on the current machine. `nil` is returned if the host is unknown.

`host` can be one of the following:

- a name string, for example "www.foobar.com"
- a dotted inet address string, for example "209.130.14.246"
- a integer representing the inet address, for example `xD1820EF6`

`fields` is a list of keywords describing what information to return for the host. If `get-host-entry` succeeds, it returns multiple values, one value for each field specified. The following fields are allowed:

- `address` The primary IP address.
- `ipv6-address` The primary IPv6 address.
- `ipv4-address` The primary IPv4 address.
- `addresses` A list of all the IP addresses.
- `ipv6-addresses` A list of all the IPv6 addresses, only.
- `ipv4-addresses` A list of all the IPv4 addresses, only.
- `name` The primary name as a string.
- `aliases` The alias names as a list of strings.
IPv4 addresses are returned as integers and IPv6 addresses are returned as objects of type `ipv6-address`.

If `ipv6` is `nil` or `t` the search is restricted to one family only IPv4 or IPv6. The default value of `ipv6` is `:any`, meaning that addresses in both families are returned. If the argument `host` is a string, that has a similar effect to using the family-specific keywords, but it may be faster. For example, these two calls returns the same addresses (possibly in a different order):

```lang-lisp
(get-host-entry "hostname"
   :fields '(:ipv6-addresses))
```

```lang-lisp
(get-host-entry "hostname"
   :fields '(:addresses) :ipv6 t)
```

If `host` is an address of the other type, that is integer with `ipv6` `t` or `ipv6-address` with `ipv6` `nil`, then `get-host-entry` first tries to do a reverse lookup to find the name of the host, and then looks for the values as if it was called with this name as the host.

When `avoid-reverse-lookup` is non-nil, `get-host-entry` avoids doing reverse lookup if `host` is a string which specifies a valid address (either IPv6 or IPv4). The default value of `avoid-reverse-lookup` is `nil`, so by default it does the lookup.

The arguments `v4mapped`, `addrconfig` and `numeric-host` have an effect only when `host` is a string. They define the flags `AI_V4MAPPED`, `AI_ADDRCONFIG` and `AI_NUMERICHOST` when doing the `getaddrinfo` call.

When `v4mapped` is `t`, the IPv6 addresses contain an IPv4 address mapped to IPv6 (`::ffff:<IPv4>`). The default value of `v4mapped` is `nil`.

When `addrconfig` is `t`, addresses of a family are returned only if the local system is configured to handle them. The default value of `addrconfig` is `nil`.

When `numeric-host` is `t`, `host` is assumed to be a numeric address, either IPv4 if dotted notation or IPv6. If it is not,
get-host-entry just returns nil. Using numerichost can speed up get-host-entry, because it prevents any DNS lookup. This has an effect only if avoid-reverse-lookup is non-nil. The default value of numerichost is nil.

Notes
1. Although the results of get-host-entry are not cached by LispWorks, the Operating System might cache them.
2. When get-host-entry is passed a string specifying an IPv6 address, the address can be followed by '%' character and a scope ID. If the scope ID is a decimal number or a valid interface name on the local system, the resulting address contains the scope ID as a number.

Examples
CL-USER 16 > (comm:get-host-entry "www.altavista.com" :fields '(:address))
3511264349
CL-USER 17 > (comm:get-host-entry 3511264349 :fields '(:name))
"altavista.com"
CL-USER 18 > (comm:get-host-entry "altavista.com" :fields '(:name :address :aliases))
"altavista.com"
3511264349
("www.altavista.com" "www.altavista.com")

get-ip-default-zone-id

Function

Summary
Gets the default zone ID of the local host.

Package
comm

Signature
get-ip-default-zone-id => result

Arguments
None.
Values  
result: An integer or a string, or nil.

Description  
The function `get-ip-default-zone-id` tries to find the local default zone ID, and if successful returns it as an integer or a string.

See also  
`ipv6-address`
`get-default-local-ipv6-address`

get-service-entry  

Function

Summary  
Returns information about a service.

Package  
comm

Signature  
`get-service-entry service protocol &key fields => value1, value2, ...`

Arguments  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>service</td>
<td>An integer or a string.</td>
</tr>
<tr>
<td>protocol</td>
<td>A string or nil.</td>
</tr>
<tr>
<td>fields</td>
<td>A list of keywords specifying which information is required.</td>
</tr>
</tbody>
</table>

Values  
`value1, value2, ...` Multiple values corresponding to the keywords in `fields`, as described below.

Description  
The function `get-service-entry` looks up `service` in the system database. If `service` is an integer, it is the port number to look up. If `service` is a string, it is a name to look up (it may be one of the aliases).

If `protocol` is a string, then `get-service-entry` looks for a system database entry with protocol `protocol`, otherwise it finds the first entry with any protocol.

`fields` specifies which information is returned. When `get-service-entry` finds an entry, it returns information about it
as multiple values corresponding to the keywords in fields. These keywords can be:

- **:name** Return the name of the entry.
- **:port** Return the port number of the entry.
- **:aliases** Return a list of aliases of the service.
- **:protocol** Return the protocol of the entry, as lowercase strings like "tcp" or "udp".

If service is an integer then the default value of fields is (:name). Otherwise the default value of fields is (:port).

**Notes**

1. **get-service-entry** tells you what the host computer knows. The results can be quite different between computers.

2. There can be multiple entries with the same name but different protocols. Many services have entries for both UDP and TCP, normally with the same port number. In many cases the protocol that is selected when you pass protocol nil is not the correct protocol to use.

**Examples**

- `(get-service-entry "smtp" nil) => 25`
- `(get-service-entry 25 nil :fields '(:name :aliases)) => "smtp", ("mail")`
- `(get-service-entry "mail" nil) => 25`

**See also**

“Specifying the target for connecting and binding a socket” on page 408

---

**get-socket-address**

**Function**

**Summary**

Returns the local address and port number of a given socket.

**Package**

comm
get-socket-address

**Signature**

get-socket-address socket => address, port

**Arguments**

socket A socket handle.

**Values**

address The local host address of the socket or nil if not connected.

port The local port number of the socket or nil if not connected.

**Description**

The function get-socket-address returns the local address of a connected socket.

**Notes**

Connected sockets have two addresses, local and remote.

**See also**

get-socket-peer-address

socket-stream-address

---

**get-socket-peer-address**

**Function**

**Summary**

Returns the remote address and port number of a given socket.

**Package**

comm

**Signature**

get-socket-peer-address socket => address, port

**Arguments**

socket A socket handle.

**Values**

address The remote host address of the socket or nil if not connected.

port The remote port number of the socket or nil if not connected.

**Description**

The function get-socket-peer-address returns the remote address of a connected socket.
Notes  Connected sockets have two addresses, local and remote.

See also  get-socket-address
          socket-stream-peer-address

get-verification-mode  
Function

Summary  Returns the mode of the SSL.

Package  comm

Signature  get-verification-mode ssl-or-ssl-ctx => result

Arguments  ssl-or-ssl-ctx  A foreign pointer of type ssl-pointer or ssl-ctx-pointer.

Values  result  A list of symbols.

Description  The function get-verification-mode returns the mode of the ssl-pointer or ssl-ctx-pointer as a list of symbols.

result is a list containing zero or more of the symbols :verify-client-once, :verify-peer and :fail-if-no-peer-cert, corresponding to the C constants VERIFY_CLIENT_ONCE VERIFY_PEER and FAIL_IF_NO_PEER_CERT respectively.

See also  set-verification-mode

ip-address-string  
Function

Summary  Returns the IP address string for an IP address. This can be either a dotted address for an integer representing an IPv4 address, or an IPv6 address string for ipv6-address.
The COMM Package

Package  comm

Signature  ip-address-string  ip-address  =>  string-ip-address

Arguments  ip-address  An integer or an ipv6-address.

Values  string-ip-address  A string, either dotted string format for an integer or an IPv6 string for ipv6-address.

Description  The function ip-address-string converts its argument to a string in the standard dotted IP address notation a.b.c.d.

The function ip-address-string converts its argument to a string in the standard IP address notation. For an IPv4 address (supplied as an integer) this is the a.b.c.d notation. For IPv6 it is the standard IPv6 address notation (not including scope ID).

See also  string-ip-address

ipv6-address

Type

Summary  Represents IPv6 addresses.

Package  comm

Description  Objects of type ipv6-address represent an IPv6 address.

ipv6-address objects are normally created by get-host-entry. They can also be created by parse-ipv6-address.

ipv6-address can be used wherever an IP address is needed, most commonly open-tcp-stream.

ipv6-address may contain a scope ID, which is not really part of the address, but is needed for using local addresses.
The string representation of an **ipv6-address** can be retrieved by **ip-address-string**. The scope ID can be accessed by **ipv6-address-scope-id**.

See also

- get-host-entry
- ipv6-address-p
- ip-address-string
- ipv6-address-scope-id
- parse-ipv6-address

### ipv6-address-p

*Function*

Summary

The predicate for objects of type **ipv6-address**.

Package

**comm**

Signature

\[ \text{ipv6-address-p} \text{ object} \Rightarrow \text{result} \]

Arguments

- **object**: A Lisp object.

Values

- **result**: A boolean.

Description

The function **ipv6-address-p** is the predicate for whether its argument **object** is of type **ipv6-address**.

See also

- **ipv6-address**

### ipv6-address-scope-id

*Function*

Summary

Returns the scope ID of an IPv6 address.

Package

**comm**

Signature

\[ \text{ipv6-address-scope-id} \text{ ipv6-address} \Rightarrow \text{scope-id} \]
The function `ipv6-address-scope-id` returns the scope ID of the IPv6 address `ipv6-address`. Global addresses have scope ID 0. `scope-id` may be a string or a number.

See also `ipv6-address`

**ipv6-address-string**

*Function*

Summary

Returns the standard string representation of an IPv6 address.

Package

`comm`

Signature

`ipv6-address-string ipv6-address => string`

Arguments

`ipv6-address` An `ipv6-address` object.

Values

`string` A string.

Description

The function `ipv6-address-string` returns the standard string representation of the address.

Notes

The result `string` does not include the scope ID.

See also `ip-address-string`

**make-ssl-ctx**

*Function*

Summary

Makes a `SSL_CTX` object.
Package | comm
---|---
Signature | make-ssl-ctx &key ssl-ctx ssl-side => ssl-ctx-ptr
Arguments | 
ssl-ctx | A symbol or a foreign pointer.
ssl-side | One of the keywords :client, :server or :both.
Values | ssl-ctx-ptr | A foreign pointer of type ssl-ctx-pointer.
Description | The function make-ssl-ctx first calls ensure-ssl, and returns a foreign pointer of type ssl-ctx-pointer.
If the value of ssl-ctx is t, :default, :v2, :v3, :v23 or :tls-v1, make-ssl-ctx creates a SSL_CTX object and returns a pointer to it.
The value of ssl-ctx can also be a foreign pointer of type ssl-ctx-pointer, in which case it is simply returned. If ssl-ctx is a foreign pointer of type ssl-pointer, then make-ssl-ctx signals an error.
The meaning of the keyword arguments ssl-ctx and ssl-side is as described for socket-stream. The default value of ssl-ctx is t and the default value of ssl-side is :server.
See also | ensure-ssl
socket-stream
ssl-ctx-pointer

loop-processing-wait-state-collection

Function
Summary | Loops processing a wait-state-collection.
Package | comm
Signature | loop-processing-wait-state-collection wait-state-collection
Arguments

\textit{wait-state-collection}

A \textit{wait-state-collection}.

Description

The function \texttt{loop-processing-wait-state-collection} loops processing \textit{wait-state-collection}.

\texttt{loop-processing-wait-state-collection} loops waiting for any state to be ready (using \texttt{wait-for-wait-state-collection}) and processes any state that is ready (using \texttt{call-wait-state-collection}). It establishes restarts that allow aborting back into the loop, and a mechanism that allows \texttt{wait-state-collection-stop-loop} to stop the loop.

If \texttt{wait-state-collection-stop-loop} is called on the \textit{wait-state-collection}, which can be from other threads, \texttt{wait-state-collection-stop-loop} stops looping and returns.

Notes

In most cases using \texttt{create-and-run-wait-state-collection} is more convenient.

There can be only one \texttt{loop-processing-wait-state-collection} on each \textit{wait-state-collection} at a time. Typically this will occur in a process that is made specifically to run \texttt{loop-processing-wait-state-collection} on the collection.

See also

\texttt{create-and-run-wait-state-collection}
\texttt{wait-for-wait-state-collection}
\texttt{wait-state-collection-stop-loop}

\textbf{make-wait-state-collection}

\textit{Function}

Summary

Returns a new empty \textit{wait-state-collection}.

Package

\texttt{comm}

Signature

\texttt{make-wait-state-collection => collection}
Values

collection

A wait-state-collection.

Description

The function make-wait-state-collection returns a new empty wait-state-collection.

See also

create-and-run-wait-state-collection

“The Async-I/O-State API” on page 412

open-tcp-stream

Function

Summary

Attempts to connect to a socket on a server and returns a stream object for the connection.

Package

comm

Signature

open-tcp-stream hostspec service &key direction element-type errorp read-timeout write-timeout timeout ssl-ctx ctx-configure-callback ssl-configure-callback handshake-timeout tlsext-host-name local-address ipv6 local-port nodelay keepalive => stream

Arguments

hostspec An integer or string or an ipv6-address object.

service A string or a fixnum.

direction One of :input, :output or :io.

element-type base-char or a subtype of integer.

errorp A boolean.

read-timeout A positive number, or nil.

write-timeout A positive number, or nil.

timeout A positive number, or nil.

ssl-ctx A symbol or a foreign pointer.
ctx-configure-callback
A function designator or \texttt{nil}. The default value is \texttt{nil}.

ssl-configure-callback
A function designator or \texttt{nil}. The default value is \texttt{nil}.

handshake-timeout
A \texttt{real} or \texttt{nil} (the default).

tlsext-host-name
A string, \texttt{t} or \texttt{nil}.

local-address \texttt{nil}, an integer, a string or a \texttt{ipv6-address} object.

ipv6 \texttt{nil}, \texttt{t} or \texttt{:any}.

local-port \texttt{nil}, a string or a fixnum.

nodelay A generalized boolean.

keepalive A generalized boolean.

Values
\texttt{stream} A \texttt{socket-stream}.

Description
The function \texttt{open-tcp-stream} attempts to connect to a socket on a server and returns \texttt{stream} for the connection if successful.

The IP address to connect to is specified by \texttt{hostspec}, and the service to provide is specified by \texttt{service}. These two arguments are interpreted as described in “Specifying the target for connecting and binding a socket” on page 408.

The direction of the connection is given by \texttt{direction}. Its default value is \texttt{:io}. The element type of the connection is determined from \texttt{element-type}, and is \texttt{base-char} by default.

If \texttt{errorp} is \texttt{nil}, failure to connect (possibly after \texttt{timeout} seconds) returns \texttt{nil}, otherwise an error is signaled.
timeout specifies a connection timeout. open-tcp-stream waits for at most timeout seconds for the TCP connection to be made. If timeout is nil it waits until the connection attempt succeeds or fails. On failure, open-tcp-stream signals an error or returns nil according to the value of errorp. To provide a timeout for reads after the connection is made, see read-timeout. The default value of timeout is nil.

read-timeout specifies the read timeout of the stream. If it is nil (the default), the stream does not time out during reads, and these may hang. See socket-stream for more details. To provide a connection timeout, see timeout.

write-timeout is similar to read-timeout, but for writes. See socket-stream for more details.

ssl-ctx, ctx-configure-callback, ssl-configure-callback and handshake-timeout are interpreted as described in “Keyword arguments for use with SSL” on page 421. Unlike the other ways of creating a socket stream with SSL processing, open-tcp-stream does not take the ssl-side argument and always uses the value :client.

If tlsext-host-name is a string, then the SNI extension in the SSL connection to set to its value. If tlsext-host-name is t and hostspec is a string, then the SNI extension in the SSL connection to set to hostspec.

If local-address is nil then the operating system chooses the local address of the socket. Otherwise the value is interpreted as for hostspec and specifies the local address of the socket. The default value of local-address is nil.

If local-port is nil then the operating system chooses the local port of the socket. Otherwise the string or fixnum value is interpreted as for service and specifies the local port of the socket. The default value of local-port is nil.

ipv6 specifies the address family to use when hostspec is a string. When ipv6 is :any, open-tcp-stream uses either of IPv4 or IPv6. When ipv6 is t, it uses only IPv6 addresses, and
when `ipv6` is `nil` it tries only IPv4. The default value of `ipv6` is `:any`.

If `keepalive` is true, `SO_KEEPALIVE` is set on the socket. The default value of `keepalive` is `nil`.

If `nodelay` is true, `TCP_NODELAY` is set on the socket. The default value of `nodelay` is `t`.

**Notes**

1. On Unix the name of the service can normally be found in `/etc/services`. If it is not there, the manual entry for services can be used to find it.

2. If `switch-open-tcp-stream-with-ssl-to-java` was called with its argument on non-nil or not supplied, when SSL-CTX is non-nil open-tcp-stream uses Java sockets instead of ordinary sockets. This is the default behavior on Android, because OpenSSL is not available on Android. The resulting streams have some limitations, most importantly `cl:listen` is not reliable on them. They also verify the host, which ordinary sockets do not currently do, in the same way that the default in `open-tcp-stream-using-java` does. See “Socket streams with Java sockets and SSL on Android” on page 432 for a full description, and `open-tcp-stream-using-java` for details about verification and which keywords are used.

**Example**

The following example opens an HTTP connection to a given host, and retrieves the root page:
(with-open-stream (http (comm:open-tcp-stream "www.lispworks.com" 80))
  (format http "GET / HTTP/1.0-C-C-C-C"
    (code-char 13) (code-char 10)
    (code-char 13) (code-char 10))
  (force-output http)
  (write-string "Waiting to reply...")
  (loop for ch = (read-char-no-hang http nil :eof)
      until ch
    do (write-char #\.)
    (sleep 0.25)
  finally (unless (eq ch :eof)
    (unread-char ch http)))
(terpri)
(loop for line = (read-line http nil nil)
      while line
    do (write-line line)))

See also

- connect-to-tcp-server
- start-up-server
- socket-stream
- socket-stream-shutdown
- switch-open-tcp-stream-with-ssl-to-java
- open-tcp-stream-using-java

"TCP and UDP socket communication and SSL" on page 407

open-tcp-stream-using-java

Function

Summary

Open a TCP stream using Java sockets for communication.

Package

comm

Signature

(open-tcp-stream-using-java hostspec service &key factory
verify direction element-type errorp read-timeout write-timeout
timeout ssl-ctx ctx-configure-callback ssl-configure-callback tlsext-host-name local-address ipv6 local-port nodelay keepalive => stream)

Arguments

- hostspec: An integer or string or an ipv6-address object.
- service: A string or a fixnum.
factory A Java socket factory.

verify t, nil, :strict, :browser-compat, a string or a JObject.

direction One of :input, :output or :io.

element-type base-char or a subtype of integer.

eror A boolean.

read-timeout A positive number, or nil.

write-timeout Ignored.

timeout A positive number, or nil.

ssl-ctx A generalized boolean.

ctx-configure-callback Ignored.

ssl-configure-callback Ignored.

tlsext-host-name Ignored.

local-address nil, an integer, a string or a ipv6-address object.

local-port nil, a string or a fixnum.

ipv6 Ignored.

nodelay A generalized boolean.

keepalive A generalized boolean.

Values stream A socket-stream.

Description The function open-tcp-stream-using-java opens a TCP stream using Java sockets for communication.
Note: `open-tcp-stream-using-java` does not have any clear advantage over `open-tcp-stream`. Use it only when you really need it.

`open-tcp-stream-using-java` accepts the same arguments as `open-tcp-stream`, plus `factory` and `verify`, but ignores the values of `write-timeout`, `ipv6`, `ctx-configure-callback`, `ssl-configure-callback` and `tlsext-host-name`. It also treats `ssl-ctx` as a generalized boolean, where any non-nil value means using SSL Java object.

`open-tcp-stream-using-java` opens and returns a `socket-stream` like `open-tcp-stream`, but the socket object that it uses is a Java object. However, `cl:listen` is unreliable on such streams, and they cannot be used in `wait-for-input-streams`. See “Socket streams with Java sockets and SSL on Android” on page 432 for details.

The keyword argument `factory` can be used to specify the socket factory to use to create the Java socket. When passed, it must be a Java socket factory, that is a `jobject` which is an instance of class `javax.net.SocketFactory`. In this case the socket is generated from this factory, the factory determines whether it is a SSL socket or not, and the value of `ssl-ctx` is used only to decide whether to do a handshake. By default, the default factory (the result of `getDefaultValue`) of `javax.net.SocketFactory` (when `ssl-ctx` is nil) or `javax.net.ssl.SSLSocketFactory` (when `ssl-ctx` is non-nil) is used.

The keyword argument `verify` is used only when `ssl-ctx` is non-nil. It controls verification of `hostspec` when SSL is used, which means checking that the certificate that was returned by the server is for this server. The default value `t` means using `SSLCertificateSocketFactory` on Android when `factory` is not supplied (see below), on other platforms it is the same as `:strict`. `:strict` mean uses the strict verifier (Java class `org.apache.http.conn.ssl.StrictHostnameVerifier`). `:browser-compat` means using "browser compatible"
The verifi er (Java class \texttt{org.apache.http.conn.ssl.BrowserCompatHostnameVerifier}). Verification with \texttt{browser-compat} is a little more relaxed than with \texttt{strict}.

On Android when \texttt{verify} is \texttt{t} and \texttt{factory} is \texttt{nil}, the code uses the socket factory \texttt{android.net.SSLCertificateSocketFactory} (instead of the default of \texttt{javax.net.ssl.SSLSocketFactory}), which is doing the verification itself. When \texttt{factory} is non-nil, Android does the same as in the previous paragraph (verify using the strict verifi er). The \texttt{SSLCertificateSocketFactory} has the advantage that it uses SNI (Server Name Indication), which makes verification work better.

When \texttt{verify} is a string, it has to be the hostname to use for verification, instead of the \texttt{hostspec} argument. The verification is done using the strict verifi er.

When \texttt{verify} is a \texttt{jobject}, it must be a verifi er (of class \texttt{javax.net.ssl.HostnameVerifier}), and it is used as-is.

The verifi er classes above are part of \texttt{httpclient} from \texttt{apache.org}, and therefore to use them (which is the default when using SSL), you need to have \texttt{httpclient}. On Android it is always available, so it is not an issue, on another architectures it needs to be added to the class path.

When \texttt{verify} is \texttt{nil}, the \texttt{hostspec} is not verified, which is not recommended. However, there are valid sites which will fail verification, because they return a certifi cate for the wrong site (that happens due to use of virtual hosts). At the time of writing, "gmail.com" is one of them, and returns a certifi cate for "mail.google.com". However, if the client uses SNI, which is used by Java socket in Java 1.7 or higher, this server does return the correct certifi cate, and in general all servers should work when using SNI. On Android the default setting uses the \texttt{SSLCertificateSocketFactory} (discussed above), which is using SNI. Thus there is a problem only when using Java 1.6 or earlier, and for Android only when you use your
own factory. For these cases, you can either use `verify nil`, or pass the name in the certificate as the `verify` argument:

```lisp
(comm:open-tcp-stream-using-java "gmail.com" 443 :ssl-ctx t :verify "mail.google.com")
```

Note however that this will fail if SNI is used.

**Notes**

1. The Java virtual machine (JVM) must be running for `open-tcp-stream-using-java` to work. On Android the JVM always runs, on other architectures it needs to have been started by `init-java-interface`. When using `ssl-ctx`, `httpclient` must be available too, and again it is always available on Android.

2. On Android, or if you call `switch-open-tcp-stream-with-ssl-to-java`, `open-tcp-stream` uses Java objects for SSL streams. The result of `open-tcp-stream` and `open-tcp-stream-using-java` with `ssl-ctx` non-nil is identical in this case.

3. Using Java sockets was added mainly for SSL streams on Android. It may be useful in other circumstances.

4. You can also make a `socket-stream` with a Java socket by passing the Java socket that your code has created to `(make-instance 'socket-stream ...)`. Note that closing such a stream will close the socket, and if you want to avoid that you need to use `replace-socket-stream-socket`.

**See also**

"Socket streams with Java sockets and SSL on Android” on page 432

`open-tcp-stream`

---

**openssl-version**

*Function*

**Summary**

Returns the version of the loaded OpenSSL library.
The function `openssl-version` returns a string specifying the version of the loaded OpenSSL library.

The argument `what` takes these values:

- **:version**
  - `result` is the version string, which looks like: "OpenSSL 0.9.7i 14 Oct 2005" or "OpenSSL 0.9.8a 11 Oct 2005"

- **:built-on**
  - Returns a string specifying when it was built.

- **:directory**
  - Returns where OpenSSL thinks it is installed.

- **:platform**
  - Returns OpenSSL's idea of which platforms it is.

- **:cflags**
  - The compilation command.

The default value of `what` is **:version**.

See also **ensure-ssl**

### parse-ipv6-address

**Function**

**Summary**

Parses a string as an IPv6 address.

**Package**

`comm`
Signature

\texttt{parse-ipv6-address \textit{string} \&\textit{key} \textit{start} \textit{end} \textit{trim-whitespace} => \textit{result}}

Arguments

\textit{string} \quad \text{A string.}

\textit{start, end} \quad \text{Bounding index designators of string.}

\textit{trim-whitespace} \quad \text{A boolean.}

Values

\textit{result} \quad \text{An \texttt{ipv6-address} object or \texttt{nil}.}

Description

The function \texttt{parse-ipv6-address} parses its argument string as an IPv6 address if possible, otherwise it returns \texttt{nil}.

\textit{start} and \textit{end} specify the subsequence of \textit{string} to parse. The default value of \textit{start} is 0. The default value of \textit{end} is \texttt{nil}, meaning the length of \textit{string}.

\textit{trim-whitespace} is a boolean specifying that leading and trailing whitespace characters may be ignored. Note that the address itself must not contain any whitespace. The default value of \textit{trim-whitespace} is \texttt{t}.

The address has to be in either standard IPv6 address notation, or dotted-quad notation. It can have the standard simplifications.

In addition, the address may be followed by a '%' character and a scope ID. If the scope ID is a string of decimal characters, it is read as a decimal number, otherwise it is taken as-is. The address may also be followed by a '/' and a prefix length in decimal format. The result \texttt{ipv6-address} object remembers the prefix length and prints it when the object is printed, but it does not affect the address otherwise.

If the syntax of the string \textit{string} is correct, \texttt{parse-ipv6-address} constructs the \texttt{ipv6-address} object and returns it. It does not perform any address resolution.

See also

\texttt{get-host-entry}

\texttt{string-ip-address}
### pem-read

**Function**

**Summary**

An interface to the SSL `PEM_read_bio_*` functions.

**Package**

`comm`

**Signature**

```lisp
pem-read thing-to-read filename &key pass-phrase callback errorp
=> result
```

**Arguments**

- `thing-to-read` A string.
- `filename` A pathname designator.
- `pass-phrase` A string, or `nil`.
- `callback` A function designator, or `nil`.
- `errorp` A generalized boolean.

**Values**

- `result` A foreign pointer or `nil`.

**Description**

The function `pem-read` is an interface to the `PEM_read_bio_*` set of functions. See the manual entry for `pem` for specifications of these functions.

`thing-to-read` defines which function is required. `pem-read` concatenates `thing-to-read` with the string "PEM_read_bio_" to form the name of the `pem` function to call.

`filename` specifies the file to load.

If `pass-phrase` is non-nil, it must be a string, which is passed to the `pem` function. The default value of `pass-phrase` is `nil`.

If `callback` is non-nil, it must be a function with signature:

```lisp
callback maximum-length rwflag => pass-phrase
```

where `maximum-length` is an integer, `rwflag` is a boolean and `pass-phrase` is the pass-phrase to use. The default value of `callback` is `nil`, but you cannot pass non-nil values for both `pass-phrase` and `callback`. 
If it succeeds, `pem-read` returns a foreign pointer to the structure that was returned by the `pem` function. If `pem-read` fails, if `errorp` is non-nil it signals an error, otherwise it returns `nil`. The default value of `errorp` is `nil`.

### read-dhparams

**Function**

**Summary**
Reads or uses cached SSL DH parameters.

**Package**
`comm`

**Signature**
```
read-dhparams filename &key pass-phrase callback errorp force => dh-ptr
```

**Arguments**
- `filename` A pathname designator.
- `pass-phrase` A string, or `nil`.
- `callback` A function designator, or `nil`.
- `errorp` A generalized boolean.
- `force` A generalized boolean.

**Values**
- `dh-ptr` A foreign pointer or `nil`.

**Description**
The function `read-dhparams` reads or uses cached DH parameters.

`filename` specifies the file to check.

Unless `force` is true, `read-dhparams` checks whether the file `filename` has already been loaded, and if it has been loaded, uses the cached value.

If `force` is true, or if there is no cached value for `filename`, `read-dhparams` loads the file by calling `pem-read` with `thing-to-read` argument "DHParams", `pass-phrase`, `callback` and `errorp`. `read-dhparams` caches and returns a foreign pointer to the
resulting DH structure (that is, a pointer corresponding to the C type `DH`).

If `read-dhparams` fails to load the file `filename`, if `errorp` is true it signals an error, otherwise it returns `nil`. The default value of `errorp` is `t`.

See also `pem-read`

**replace-socket-stream-socket**

**Function**

**Summary**

Replaces the socket in a `socket-stream`, returning the existing socket object without closing it.

**Package**

`comm`

**Signature**

`replace-socket-stream-socket socket-stream socket => socket-or-nil`

**Arguments**

- `socket-stream` A `socket-stream`.
- `socket` A socket object or `nil`.

**Values**

- `socket-or-nil` A socket object or `nil`.

**Description**

The function `replace-socket-stream-socket` replaces the socket in the `socket-stream socket-stream`, returning the existing socket object without closing it.

A socket object is normally a socket in the native operating system sense (an integer representing an `fd` socket on Unix, and an integer representing `SOCKET` on Microsoft Windows), but when using the Java interface it can also be a Java socket (`jobject` of class `java.net.Socket`).

`replace-socket-stream-socket` sets the socket in `socket-stream` to the argument `socket`, and then returns the old socket object without closing it.
Notes

1. Getting the old socket using the `socket-stream` accessor `socket-stream-socket` and then using `(setf socket-stream-socket)` to set the new one is different, because `cl:setf` will close the old socket.

2. Passing `nil` as the socket allows you to close the stream while retaining the socket.

3. The new socket does not need to be the same kind of socket as the old one.

See also `socket-stream`

**server-terminate**

*Function*

**Summary**
Terminates a server.

**Package**
`comm`

**Signature**
`server-terminate &optional process => result`

**Arguments**
`process` A `mp:process` object or `nil`.

**Values**
`result` A boolean.

**Description**
The function `server-terminate` terminates a server process.

If `process` is a process object it must be the result of a call to `start-up-server`. `server-terminate` terminates it, and frees all the associated resources.

If `process` is `nil` or is not supplied, the call to `server-terminate` must be inside the scope of the process that was created by `start-up-server`, which can by either `function` or `announce` that you passed to `start-up-server`. `server-terminate` returns `t` in this case, and the actual termination happens after your function (that is, `function` or `announce`) returns.
server-terminate returns t if the server was still active when it was called, otherwise it returns nil. It can be called repeatedly on the same server, and can be used as a predicate to check whether the server really went away.

Notes

In LispWorks 6.0 and earlier versions, process-kill is the way to terminate servers. This is deprecated, because it may leave some value in an invalid state.

See also

start-up-server

set-verification-mode

Function

Summary

Sets the verification mode for CTX.

Package comm

Signature set-verification-mode ssl-ctx ssl-side mode &optional callback

Arguments

ssl-ctx A foreign pointer of type ssl-pointer or ssl-ctx-pointer.

ssl-side :server or :client.

mode An integer, one of the symbols :never, :always, :once, or a list of keywords.

callback A foreign function.

Values

result A list of symbols.

Description

The function set-verification-mode sets the verification mode for CTX according to arguments ssl-side and mode.

When ssl-side is :server, mode can be:

An integer mode is passed directly to SSL_set_verify or SSL_CTX_set_verify.
The server will not send a client certificate request to the client, so the client will not send a certificate.

The server sends a client certificate request to the client. The certificate returned (if any) is checked. If the verification process fails, the TLS/SSL handshake is immediately terminated with an alert message containing the reason for the verification failure.

Same as :always except that the client certificate is checked only on the initial TLS/SSL handshake, and not again in case of renegotiation.

The list contains (some of) the keywords :verify-client-once, :verify-peer and :fail-if-no-peer-cert. These keywords map to the corresponding C constants VERIFY_CLIENT_ONCE, VERIFY_PEER and FAIL_IF_NO_PEER_CERT respectively. See the manual entry for SSL_CTX_set_verify for the meaning of the constants.

When ssl-side is :client, mode can be:

An integer mode is passed directly as for ssl-side :server.

If not using an anonymous cipher, the server will send a certificate which will be checked by the client. The handshake will be continued regardless of the verification result.

The server certificate is verified. If the verification process fails, the TLS/SSL handshake is immediately terminated with an alert message containing the reason for the verifi-
cation failure. If no server certificate is sent because an anonymous cipher is used, verification is ignored.

A list The list contains keywords as described above for *ssl-side* :server.

If non-nil *callback* should be a symbol, function, string or foreign pointer designating a foreign function that is called to perform verification. The default value of *callback* is nil.

See also get-verification-mode

**set-ssl-ctx-dh**

*Function*

**Summary** Sets the DH parameters for a *SSL_CTX*.

**Package** comm

**Signature**

```
set-ssl-ctx-dh ssl-ctx &key dh filename func filename-list passphrase callback => result
```

**Arguments**

- *ssl-ctx* A foreign pointer.
- *filename* A pathname designator or nil.
- *func* A function designator or nil.
- *filename-list* An association list.
- *passphrase* A string, or nil.
- *callback* A function designator, or nil.

**Values**

- *result* A boolean.

**Description**

The function *set-ssl-ctx-dh* sets the DH parameters for a *SSL_CTX*.

*ssl-ctx* can be either a foreign pointer of type *ssl-ctx-pointer* or a foreign pointer of type *ssl-pointer*. 
The value is to use is specified by one of the parameters \(dh\), \textit{filename}, \textit{func} or \textit{filename-list}.

If \(dh\) is non-nil, it must be a foreign pointer to a DH (corresponding to the C type \texttt{DH*}), and this DH is used as-is. The default value of \(dh\) is \texttt{nil}.

Otherwise, if \textit{filename} is non-nil, it must be a pathname designator for a file containing DH parameters, which is loaded (by \texttt{read-dhparams}) and then used. In this case, \textit{pass-phrase} and \textit{callback} can be used, and are passed to \texttt{pem-read}.

Otherwise, if \textit{func} is non-nil, it must be a function with signature:

\[
\text{func is-export keylength} \Rightarrow dh\text{-ptr}
\]

where \textit{is-export} is a boolean, \textit{keylength} is an integer, and \textit{dh\text{-ptr}} is a pointer to an appropriate DH structure. \texttt{set-ssl-ctx-dh} installs \textit{func} as the DH callback.

Otherwise (that is, if each of \(dh\), \textit{filename} and \textit{func} are \texttt{nil}) then \textit{filename-list} must be a non-nil association list of keylengths and filenames, sorted by the keylengths in ascending order (that is, larger keylengths are towards the end of the list). \texttt{set-ssl-ctx-dh} installs a DH callback which when called finds the first keylength which is equal or bigger than the required keylength, loads the associated file (by calling \texttt{read-dhparams}), and returns it. It also loads the first file of the list immediately.

\textit{result} is \texttt{t} on success, \texttt{nil} otherwise.

See also \texttt{pem-read}\newline
\texttt{read-dhparams}\newline
\texttt{ssl-ctx-pointer}\newline
\texttt{ssl-pointer}
set-ssl-ctx-options  Function

Summary  Sets the options in a SSL_CTX.

Package  comm

Signature  set-ssl-ctx-options ssl-ctx &key microsoft_sess_id_bug netscape_challenge_bug netscape_reuse_cipher_change_bug sslref2_reuse_cert_type_bug microsoft_big_sslv3_buffer msie_sslv2_rsa_padding ssleay_080_client_dh_bug tls_d5_bug tls_block_padding_bug dont_insert_empty_segments all no_session_resumption_on_renegotiation single_dh_use ephemeral_rsa cipher_server_preference tls_rollback_bug no_sslv2 no_sslv3 no_tlsv1 pkcs1_check_1 pkcs1_check_2 netscape_ca_dn_bug netscape_demo_cipher_change_bug

Arguments  

ssl-ctx  A foreign pointer.

Each of the keyword arguments is a generalized boolean defaulting to nil.

Description  The function set-ssl-ctx-options sets the options in a SSL_CTX.

ssl-ctx can be either a foreign pointer of type ssl-ctx-pointer or a foreign pointer of type ssl-pointer.

The option that is set is the logior of all the options that are passed to set-ssl-ctx-options via the keyword arguments. The value used for each non-nil keyword keyword is the value of SSL_OP_keyword. The meaning of the options is specified in the OpenSSL manual page for SSL_set_options.

See also  ssl-ctx-pointer
ssl-pointer

set-ssl-ctx-password-callback  Function

Summary  Sets the password for a SSL_CTX.
Package: comm

Signature: set-ssl-ctx-password-callback ssl-ctx &key callback password

Arguments:
- ssl-ctx: A foreign pointer.
- callback: A function designator, or nil.
- password: A string, or nil.

Description: The function set-ssl-ctx-password-callback sets the password for a SSL_CTX, either to a callback or a password. 

ssl-ctx should be a foreign pointer of type ssl-ctx-pointer.

If callback is non-nil, it must be a function with signature:

\[
\text{callback} \quad \text{maximum-length} \quad \text{rwflag} \Rightarrow \text{result}
\]

where maximum-length is an integer, rwflag is a boolean and result is a string. The default value of callback is nil.

If password is non-nil and callback is nil, a callback is installed that simply returns password. The default value of password is nil.

If both callback and password are nil, set-ssl-ctx-password-callback signals an error.

See also: ssl-ctx-pointer

---

**set-ssl-library-path**

Function

Summary: Sets the SSL library path.

Package: comm

Signature: set-ssl-library-path library-path

Arguments:
- library-path: A string or a list of strings.
The function `set-ssl-library-path` sets the SSL library path.

`library-path` should be a string or a list of strings. Each string specifies a library to load. The libraries are loaded in the order they are in the list.

Note that in contrast to `ensure-ssl`, the effect of `set-ssl-library-path` persists after saving and restarting the image.

See also

`ensure-ssl`  
“How LispWorks locates the OpenSSL libraries” on page 420

### socket-error

**Class**

**Summary**  
The condition class for socket errors.

**Package**  
`comm`

**Superclasses**  
`simple-error`

**Subclasses**  
`ssl-condition`

**Initargs**  
`:stream`  
A socket-stream.

**Description**  
The condition class for socket errors.

### socket-stream

**Class**

**Summary**  
The socket stream class.

**Package**  
`comm`

**Superclasses**  
`buffered-stream`

**Initargs**  
`:socket`  
A socket handle.
:direction One of :input, :output, or :io.
:element-type One of base-char, (signed-byte 8) and (unsigned-byte 8).
:read-timeout A positive number or nil.
:write-timeout A positive number or nil.
:ssl-ctx A keyword, t or nil, or a foreign pointer of type ssl-ctx-pointer or ssl-pointer.
:ssl-side One of the keywords :client, :server or :both. The default value is :server.
:ctx-configure-callback A function designator or nil.
:ssl-configure-callback A function designator or nil.
:handshake-timeout A real or nil (the default).
:tlsext-host-name A string or nil.

Accessors

socket-stream-socket
stream:stream-read-timeout
stream:stream-write-timeout

Description

The class socket-stream implements a buffered stream connected to a socket. The socket handle, specified by :socket, and the direction, specified by :direction, must be passed for a meaningful stream to be constructed. Common Lisp input functions such as read-char will see end-of-file if the other end of the socket is closed.

The :element-type initarg determines the expected element type of the stream traffic. However, stream input and output functions for character and binary data generally work in the
obvious way on a socket-stream with any of the allowed values of element-type. For example, read-sequence can be called with a string buffer and a binary socket-stream: the character data is constructed from the input as if by code-char. Similarly write-sequence can be called with a string buffer and a binary socket-stream: the output is converted from the character data as if by char-code. Also, 8-bit binary data can be read and written to a base-char socket-stream.

All standard stream I/O functions except for write-byte and read-byte have this flexibility.

The :read-timeout initarg specifies the read timeout in seconds, or is nil, meaning there are no timeouts during reads (this is the default).

The read-timeout property is intended for use when a socket connection might hang during a call to any Common Lisp input function. The read-timeout can be set by make-instance or by open-tcp-stream. It can also be modified by (setf (stream:stream-read-timeout)). When read-timeout is nil, there is no timeout during reads and the call may hang. When read-timeout is not nil, and there is no input from the socket for more than read-timeout seconds, any reading function returns end-of-file. The read-timeout does not limit the time inside read, but the time between successful extractions of data from the socket. Therefore, if the reading needs several rounds it may take longer than read-timeout.

Using (setf (stream:stream-read-timeout)) on the stream while it is inside a read function has undefined effects. However, the setf function can be used between calls to read functions. The read-timeout property of a stream can be read by (stream:stream-read-timeout stream).

The :write-timeout initarg specifies the write timeout in seconds, or is nil, meaning that there are no timeouts during writes (this is the default).
The write-timeout property is similar to read-timeout, but for write operations. If flushing the stream buffer takes too long then error is called.

The initargs :ssl-ctx, :ssl-side, :ctx-configure-callback, :ssl-configure-callback and :handshake-timeout can be be supplied to create and configure socket streams with SSL processing. See “Keyword arguments for use with SSL” on page 421 for more details.

If :tlsexit-host-name initarg is a string then the SNI extension in the SSL connection to set to its value.

If there is a non-local exit while initializing the socket-stream (the most common reason being a SSL handshake failure when using SSL), then the stream will be closed. This will cause the socket to be closed as well.

Notes

1. The function wait-for-input-streams and wait-for-input-streams-returning-first are a convenient interface for waiting for input from socket streams. The standard I/O functions (cl:read, cl:read-char and so on) can also wait properly. You can also use process-wait and similar functions with cl:listen in the wait-function, but you will need to use with-noticed-socket-stream.

2. The socket object in a socket-stream is normally a socket object in the operating system sense. On Unix and Microsoft Windows it is an integer corresponding to a socket as returned from the C functions socket and accept. It can also be a Java socket object, see “Socket streams with Java sockets and SSL on Android” on page 432 for details.

3. (setf socket-stream-socket) can be used to set the socket object in the stream, and can also set it to nil. When there is already a socket in the stream, (setf socket-stream-socket) closes it before setting the slot to
the new socket. The function \texttt{replace-socket-stream-socket} can be used to set the socket without closing the old one.

**Example**

The following makes a bidirectional stream connected to a socket specified by \texttt{handle}.

\begin{verbatim}
(make-instance 'comm:socket-stream
  :socket 'handle
  :direction :io
  :element-type 'base-char)
\end{verbatim}

This example creates a socket stream with a read-timeout:

\begin{verbatim}
(make-instance 'comm:socket-stream
  :handle 'handle
  :direction :input
  :read-timeout 42)
\end{verbatim}

The following form illustrates character I/O in a binary \texttt{socket-stream}:

\begin{verbatim}
(with-open-stream (x
  (comm:open-tcp-stream
    "localhost" 80
    :element-type '(unsigned-byte 8)))
  (write-sequence (format nil "GET / HTTP/1.0~%~%") x)
  (force-output x)
  (let ((res (make-array 20 :element-type 'base-char)))
    (values (read-sequence res x) res)))
\end{verbatim}

The following form illustrates binary I/O in a \texttt{base-char} \texttt{socket-stream}:

\begin{verbatim}
(with-open-stream (x
  (comm:open-tcp-stream
    "localhost" 80
    :element-type '(unsigned-byte 8)))
  (write-sequence (format nil "GET / HTTP/1.0~%~%") x)
  (force-output x)
  (let ((res (make-array 20 :element-type 'base-char)))
    (values (read-sequence res x) res)))
\end{verbatim}
(with-open-stream (x
  (comm:open-tcp-stream
    "localhost" 80
    :element-type 'base-char))
  (write-sequence
    (map '(simple-array (unsigned-byte 8) 1)
      'char-code
      (format nil "GET / HTTP/1.0~%~%"))
    x)
  (force-output x)
  (let ((res (make-array 20
               :element-type
               '(unsigned-byte 8)))))
  (values (read-sequence res x)
           (map 'string 'code-char res))))

See also
connect-to-tcp-server
open-tcp-stream
start-up-server
stream-read-timeout
wait-for-input-streams
replace-socket-stream-socket
“TCP and UDP socket communication and SSL” on page 407

socket-stream-address

Function

Summary
Returns the local address and port number of a given socket stream.

Package
comm

Signature
socket-stream-address stream => address, port

Arguments
stream A socket stream.

Values
address The local host address of the socket stream or nil if not connected.
port The local port number of the socket stream or nil if not connected.
Description  The function `socket-stream-address` returns the local address of a connected socket.

Notes  Connected socket streams have two addresses, local and remote,

See also  `socket-stream-peer-address`
          `get-socket-address`

---

`socket-stream-ctx`  

Function

**Summary**  
Accesses the `SSL_CTX` attached to a socket stream.

**Package**  `comm`

**Signature**  
`socket-stream-ctx socket-stream => ssl-ctx-pointer`

**Arguments**  
`socket-stream`  A `socket-stream`.

**Values**  
`ssl-ctx-pointer`  A foreign pointer of type `ssl-ctx-pointer`, or `nil`.

**Description**  
The function `socket-stream-ctx` accesses the `SSL_CTX` that is attached to the `socket-stream socket-stream`. It returns `nil` if SSL is not attached.

See also  
`socket-stream`
          `ssl-ctx-pointer`

---

`socket-stream-handshake`  

Function

**Summary**  
Perform a SSL handshake on a stream.

**Package**  `comm`
Signature  
socket-stream-handshake stream &optional timeout => success

Arguments  
stream  A socket-stream.
timeout  nil or a real.

Values  
success  A boolean.

Description  
The function socket-stream-handshake performs a handshake on stream, which must be attached to SSL.

socket-stream-handshake returns false if the handshake does not finish in timeout seconds or if the SSL connection was cleanly closed by the other side. Other failures cause an error to be signaled.

socket-stream-handshake returns true on success.

Notes  
The other socket-stream interface functions signal errors if the handshake fail for any reason, including timeout or clean close.

If SSL was attached with ssl-side :both, then you will need to specify which side to take in the handshake by calling ssl-set-accept-state or ssl-set-connect-state with the ssl-pointer return by socket-stream-ssl.

See also  
socket-stream
“Using SSL” on page 418

socket-stream-peer-address  
Function  

Summary  
Returns the remote address and port number of a given socket stream.

Package  comm

Signature  
socket-stream-peer-address stream => address, port
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream</code></td>
<td>A socket-stream.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>address</code></td>
<td>The remote host address of the socket stream or <code>nil</code> if not connected.</td>
</tr>
<tr>
<td><code>port</code></td>
<td>The remote port number of the socket stream or <code>nil</code> if not connected.</td>
</tr>
</tbody>
</table>

Description

Connected socket streams have two addresses, local and remote. The function `socket-stream-peer-address` returns the remote address.

See also

- `socket-stream-address`
- `get-socket-peer-address`

socket-stream-shutdown

Function

Summary

Performs a shutdown on one or both sides of a TCP socket connection.

Package

comm

Signature

`socket-stream-shutdown stream direction &key abort`

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream</code></td>
<td>A socket-stream.</td>
</tr>
<tr>
<td><code>direction</code></td>
<td>One of <code>:input</code>, <code>:output</code> or <code>:io</code>.</td>
</tr>
<tr>
<td><code>abort</code></td>
<td>A generalized boolean.</td>
</tr>
</tbody>
</table>

Description

The function `socket-stream-shutdown` performs a shutdown on one or both sides of a TCP socket connection of `stream`, which can indicate to the peer that no more data will be sent or received.

When `direction` is `:input`, receive operations are shut down.
When `direction` is `:output`, send operations are shut down.
When `direction` is `:io`, all operations are shut down.
If `abort` is true and `direction` is `:output` or `:io`, then any input or output in the socket stream buffers is discarded. Otherwise output is flushed and input is left in the buffer.

It is an error to read from `stream` (after no data is left in the buffer) after shutdown for `:input` or `:io` or to write to `stream` after shutdown for `:output` or `:io`.

Notes  

`socket-stream-shutdown` does not close the socket stream, so it is still necessary to call `close` to free resources associated with the stream.

See also  

`socket-stream`  

socket-stream-ssl  

**Function**

**Summary**  
Accesses the SSL attached to a socket stream.

**Package**  
`comm`

**Signature**  
`socket-stream-ssl socket-stream => ssl-pointer`

**Arguments**  
`socket-stream`  
A socket-stream.

**Values**  
`ssl-pointer`  
A foreign pointer of type `ssl-pointer`, or `nil`.

**Description**  
The function `socket-stream-ssl` accesses the SSL that is attached to the socket-stream `socket-stream`. It returns `nil` if SSL is not attached.

**See also**  
`socket-stream`  
`ssl-pointer`
**socket-stream-ssl-side**

**Function**

**Summary**  
Accesses the ssl-side of a socket stream.

**Package**  
comm

**Signature**  
socket-stream-ssl-side socket-stream => ssl-side

**Arguments**  
socket-stream  
A socket-stream.

**Values**  
ssl-side  
:client, :server, :both or nil.

**Description**  
The function socket-stream-ssl-side accesses the ssl-side of the socket-stream. It returns nil if SSL is not attached.

**Note**  
socket-stream-ssl-side is useful as a predicate for testing if an socket-stream has SSL attached.

**See also**  
socket-stream

---

**ssl-cipher-pointer**

**FLI type descriptor**

**Summary**  
An FLI type for use with SSL.

**Package**  
comm

**Signature**  
ssl-cipher-pointer

**Description**  
The FLI type ssl-cipher-pointer corresponds to the C type SSL_CIPHER*.

---

**ssl-cipher-pointer-stack**

**FLI type descriptor**

**Summary**  
An FLI type for use with SSL.
ssl-cipher-pointer-stack

Description
The FLI type `ssl-cipher-pointer-stack` corresponds to the C type `STACK_OF(SSL_CIPHER)`.

ssl-closed

Class

Summary
The class for SSL errors corresponding to `SSL_ERROR_ZERO_RETURN`.

Package
`comm`

Superclasses
`ssl-condition`

Description
The condition class `ssl-closed` corresponds to `SSL_ERROR_ZERO_RETURN`. It means the underlying socket is dead.

ssl-condition

Class

Summary
The condition class for SSL errors.

Package
`comm`

Superclasses
`socket-error`

Subclasses
`ssl-closed`
`ssl-error`
`ssl-failure`
`ssl-x509-lookup`

Description
The condition class for errors inside SSL.
**ssl-ctx-pointer**

*FLI type descriptor*

**Summary**
An FLI type for use with SSL.

**Package**
comm

**Signature**
ssl-ctx-pointer

**Description**
The FLI type `ssl-ctx-pointer` corresponds to the C type `SSL_CTX*`.

---

**ssl-error**

*Class*

**Summary**
The class for SSL errors corresponding to `SSL_ERROR_SYSCALL`.

**Package**
comm

**Superclasses**
ssl-condition

**Description**
The condition class `ssl-error` corresponds to `SSL_ERROR_SYSCALL`. It means that something got broken.

---

**ssl-failure**

*Class*

**Summary**
The class for SSL errors corresponding to `SSL_ERROR_SSL`.

**Package**
comm

**Superclasses**
ssl-condition

**Description**
The condition class `ssl-failure` corresponds to `SSL_ERROR_SSL`. This means a failure in processing the input, typically due to a mismatch between the client and the server.
You get this error when trying to use a SSL connection to a non-secure peer.

**ssl-new**

**Function**

Summary

Creates a SSL.

Package

comm

Signature

`ssl-new ssl-ctx-pointer => ssl-pointer`

Arguments

`ssl-ctx-pointer` A foreign pointer of type `ssl-ctx-pointer`.

Values

`ssl-pointer` A foreign pointer of type `ssl-pointer`.

Description

The function `ssl-new` creates a SSL by a direct call to the C function `SSL_new`. It returns a pointer to the new SSL.

See also

`ssl-ctx-pointer`

`ssl-pointer`

**ssl-pointer**

**FLI type descriptor**

Summary

An FLI type for use with SSL.

Package

comm

Signature

`ssl-pointer`

Description

The FLI type `ssl-pointer` corresponds to the C type `SSL*`. 
ssl-x509-lookup

Class

Summary
The class for SSL errors corresponding to
SSL_ERROR_WANT_X509_LOOKUP.

Package
comm

Superclasses
ssl-condition

Description
The condition class ssl-x509-lookup corresponds to
SSL_ERROR_WANT_X509_LOOKUP. It happens when a certifi-
cate is rejected by a user callback.

start-up-server

Function

Summary
Starts a TCP server.

Package
comm

Signature
start-up-server &key function announce service address local-
address local-port nodelay keepalive process-name wait error =>
process, startup-condition

Arguments
function A function name.
announce An output stream, t, nil or a function.
service An integer, a string or nil.
backlog nil or a positive integer.
address A synonym for local-address.
local-address An integer, an ipv6-address object, a string
or nil.
local-port A synonym for service.
nodelay A generalized boolean.
keepalive A generalized boolean.
**Values**

- **process**: A process, or `nil`.
- **startup-condition**: A condition object, or `nil`.

**Description**

The function **start-up-server** starts a TCP server. Use `open-tcp-stream` to send messages from another client to the server.

The function argument provides the name of the function that processes connections. When a connection is made, the function is called with the connected socket handle, at which point you can make a stream using `make-instance` and communicate with the client. The server does not accept more connections until the function returns, so normally it should create another light-weight process to handle the connection. However, the operating system typically provides a small queue of partially accepted connections, which prevents connection failure for new clients until the server is ready to accept more connections. If the function is not specified, the built-in Lisp listener server is used. See the examples section below.

If `announce` is a stream or `t` (denoting `*standard-output*`), a message appears on the stream when the server is started.

If `announce` is a function it is called when the server is started. `announce` should take two arguments: `socket` and `condition`. `socket` is the socket used by the server. `announce` can therefore be used to record this socket. `condition` describes the error if there is one. `announce` can be called with `socket nil` and a condition only if `error` is `nil`. If the process is killed, `announce` is called with `socket nil` and `condition nil`.

The default for `announce` is `nil`, meaning there is no message.
service is interpreted as described in “Specifying the target for connecting and binding a socket” on page 408. The default value of service is a string "lispworks".

backlog specifies the maximum number of pending connections for the socket in the operating system (see your operating system's documentation for the function listen). The default value of backlog is 5.

If local-address is nil then the server will receive connections to all IP addresses on the computer. If local-address is non-nil then the server only receives connections for the IP address that local-address specifies. The default value of local-address is nil.

address also determines which family is used when making the socket. AF_INET6 is used in these cases:

- The address is an ipv6-address.
- The address is a string specifying an IPv6 address.
- The address is a string that resolves to an IPv6 address.

Otherwise AF_INET is used. When address is not supplied, AF_INET is used. To open a server with AF_INET6 listening to any address, either use the keyword argument ipv6 or pass the zero IPv6 address "::".

If keepalive is true, SO_KEEPALIVE is set on the socket. The default value of keepalive is nil.

If nodelay is true, TCP_NODELAY is set on the socket. The default value of nodelay is t.

The process-name specifies the process name. The default is constructed from the service name in the following fashion:

(format nil "~S server" service)

The wait argument controls whether start-up-server waits for the server to start or returns immediately. When wait is non-nil and an error was signaled, process is nil and the error is returned in startup-condition Otherwise just one
value, the server process, is returned. The default for wait is nil.

ipv6 affects the resolution of address if it is a string or nil. When ipv6 is nil, it forces IPv4 addresses, and if ipv6 is t it forces IPv6 addresses. When ipv6 is :any the system tries either IPv4 or IPv6 and uses the first socket that it succeeds to bind. When ipv6 is :both the system uses IPv6 (like the value t) but allows connection requests in IPv4. Note that with t only IPv6 connections are allowed. The default value of ipv6 is :any.

The error argument controls what happens if an error is signaled in the server thread. If error is nil then the thread is terminated. If error is non-nil then the debugger is entered. The default value for error is (not wait).

Notes
1. Some versions of Microsoft Windows fail to detect the case where more than one server binds a given port, so an error will not be raised in this situation.
2. When the server is not needed any more, terminate it by calling server-terminate with the process returned by start-up-server as its argument, or call server-terminate from the function supplied to start-up-server.
3. When using using ipv6 t, it is possible to listen separately for IPv4 connections on the same service (by another service or using the Asynchronous I/O API). When using :both, it is not possible to listen separately to IPv4 on the same service.
4. The server has a mechanism that checks for repeated unexplained failures associated with accepting sockets, and if that happens too often it closes the accepting socket and opens it again. When that happens, announce is called again with the same arguments. If service was nil, the port that the underlying system assigned to the first socket is used for opening the socket again. One situation
that invokes that mechanism is putting an iOS device to sleep, which causes the accepting socket to become broken in a non-obvious way.

Compatibility note

In LispWorks 6.1 and previous versions, the argument `ipv6 t` means either accepting IPv4 or not, depending on the default of the operating system. In LispWorks 7.0 and later `ipv6 t` means never allow IPv4 connections.

Examples

The following example uses the built-in Lisp listener server:

```lisp
(comm:start-up-server :service 10243)
```

It makes a Lisp listener server on port 10243 (check with local network managers that this port number is safe to use). When a client connects to this, Lisp calls `read`. The client should send a string using Common Lisp syntax followed by a newline. This string is used to name a new light-weight process that runs a Lisp listener. When this has been created, the server waits for more connections.

The next example illustrates the use of the `function` argument. For each line of input read by the server it writes the line back with a message. The stream generates `EOF` if the other end closes the connection.

```lisp
(defvar *talk-port* 10244) ; a free TCP port number

(defun make-stream-and-talk (handle)
  (let ((stream (make-instance 'comm:socket-stream
                               :socket handle
                               :direction :io
                               :element-type 'base-char)))
    (mp:process-run-function (format nil "talk ~D" handle)
                              '()
                              'talk-on-stream stream)))
```
(defun talk-on-stream (stream)
  (unwind-protect
      (loop for line = (read-line stream nil nil)
           while line
           do
           (format stream "You sent: ‘~A’~\n% line)
           (force-output stream))
     (close stream)))

(comm:start-up-server :function ’make-stream-and-talk
 :service *talk-port*)

This is a client which uses the talk server:

(defun talking-to-myself ()
  (with-open-stream
    (talk (comm:open-tcp-stream "localhost" *talk-port*))
    (dolist (monolog
             ’("Hello self.
             "Why don’t you say something original?"
             "Talk to you later then.  Bye."
             ))
      (write-line monolog talk)
      (force-output talk)
      (format t "I said: \"~A\"\n" monolog)
      (format t "Self replied: \"~A\"\n" (read-line talk nil nil)))))

(talking-to-myself)
=>
I said: "Hello self."
Self replied: "You sent: 'Hello self.'"
I said: "Why don’t you say something original?"
Self replied: "You sent: 'Why don’t you say something original?'
I said: "Talk to you later then.  Bye."
Self replied: "You sent: 'Talk to you later then. Bye.'"

This example illustrates a server which picks a free port and
records the socket. The last form queries the socket for the
port used.
(defvar *my-socket* nil)

(defun my-announce-function (socket condition)
  (if socket
      (setf *my-socket* socket)
      (my-log-error condition)))

(comm:start-up-server :service nil
 :error nil
 :announce 'my-announce-function)

(multiple-value-bind (address port)
    (comm:get-socket-address *my-socket*)
    port)

See also
open-tcp-stream
server-terminate
socket-stream

start-up-server-and-mp

Function

Package
comm

Signature
start-up-server-and-mp &key function announce service local-address address process-name

Arguments
function A function name.
announce An output stream, t, nil or a function.
service An integer, a string or nil.
local-address An integer, a string or nil.
address A synonym for local-address.
process-name A symbol or expression.

Description
The function start-up-server-and-mp starts multiprocessing if it has not already been started and then calls start-up-server with the supplied function, announce, service, local-address and process-name arguments.
Notes  \textit{start-up-server-and-mp} is not implemented on Microsoft Windows.

See also  \textit{start-up-server}

\textbf{string-ip-address} \quad \textit{Function}

Summary  Returns either an integer representing an IPv4 address or an \texttt{ipv6-address} object from the given IP address string.

Package  \texttt{comm}

Signature  \texttt{string-ip-address \ ip-address-string => ip-address}

Arguments  \texttt{ip-address-string}  A string denoting an IP address in either dotted format for IPv4 or standard IPv6 format.

Values  \texttt{ip-address}  Either an integer representing an IPv4 address, or an \texttt{ipv6-address} object.

Description  The function \texttt{string-ip-address} takes a string in the standard dotted IP address notation \texttt{a.b.c.d} and returns the corresponding integer IP address.

The function \texttt{string-ip-address} takes a string and tries to parse as an IP address. If \texttt{ip-address-string} is in a proper dotted IP address format, it returns an integer representing an IPv4 address. Otherwise it tries to read it as an IPv6 address using \texttt{parse-ipv6-address} (with \texttt{trim-whitespace-p nil}), which returns an \texttt{ipv6-address} object if it is successful or \texttt{nil} if it fails.

See also  \texttt{ip-address-string}
\texttt{parse-ipv6-address}
**switch-open-tcp-stream-with-ssl-to-java**

*Function*

**Summary**
Make `open-tcp-stream` use Java sockets for SSL streams.

**Package**
`comm`

**Signature**
```
switch-open-tcp-stream-with-ssl-to-java &optional on
```

**Arguments**
on
A generalized boolean.

**Description**
The function `switch-open-tcp-stream-with-ssl-to-java` makes `open-tcp-stream` use Java sockets for SSL streams.

The default state corresponds to `on` being `nil`, except on Android when `switch-open-tcp-stream-with-ssl-to-java` is called before delivering to Android (if the module "comm" was loaded) to switch the state to `t`. The default value of `on` is `t`.

Once the state switches to `t`, when `open-tcp-stream` is called with `ssl-ctx` non-nil, it uses a Java socket instead of ordinary socket to implement the stream. The resulting stream has some limitations, in particular `cl:listen` does not work reliably on it. See “Socket streams with Java sockets and SSL on Android” on page 432 for details.

**Notes**
1. The Java virtual machine (JVM) must be running for `open-tcp-stream` to work after it is switched to use Java sockets. On Android the JVM always runs, on other architectures it needs to have been started by `init-java-interface`.

2. `open-tcp-stream-using-java` can be used to make plain (non-SSL) socket streams with Java sockets, if that seems to be useful.

**See also**
`open-tcp-stream`
`open-tcp-stream-using-java`
Function

wait-for-wait-state-collection

Summary
Waits for a state in a collection to become active.

Package
comm

Signature
wait-for-wait-state-collection collection

Arguments
collection A wait-state-collection.

Description
The function wait-for-wait-state-collection waits for one of the states in collection to become active, or until some message arrives from another process. Such messages may be a result of creating a new async-io-state associated with the collection, or a result of a call to apply-in-wait-state-collection-process. wait-for-wait-state-collection returns once any of the states in collection is ready or there is a message.

Notes
Typically you would not call wait-for-wait-state-collection yourself, but it will be called by loop-processing-wait-state-collection. However, sometimes you may want to create the looping code yourself. In the latter case, once wait-for-wait-state-collection returns, you will need to call call-wait-state-collection to handle the active states or messages in collection.

You can use apply-in-wait-state-collection-process with a function that does nothing (e.g. false) to wake up a waiting call to wait-for-wait-state-collection on a specific collection.
See also create-and-run-wait-state-collection
loop-processing-wait-state-collection
“The Async-I/O-State API” on page 412

wait-state-collection

Class

Summary An object that controls asynchronous I/O via an event loop.

Package comm

Superclasses None

Subclasses None

Description Objects of class wait-state-collection control asynchronous I/O via an event loop.

See also “The Async-I/O-State API” on page 412

wait-state-collection-stop-loop

Function

Summary Stops a loop which is processing a wait-state-collection.

Package comm

Signature wait-state-collection-stop-loop wait-state-collection

Arguments wait-state-collection

A wait-state-collection.

Description The function wait-state-collection-stop-loop stops a loop which is processing wait-state-collection.
If there is currently a call to loop-processing-wait-state-collection with the wait-state-collection, wait-state-collection-stop-loop makes it stop and return.

Notes

wait-state-collection-stop-loop can be called from any process.

See also

loop-processing-wait-state-collection

---

**with-noticed-socket-stream**

*Macro*

**Package**

comm

**Signature**

with-noticed-socket-stream (stream) &body body

**Arguments**

- stream: A stream created using open-tcp-stream.
- body: Code to be executed while the stream is “noticed”.

**Description**

The macro with-noticed-socket-stream evaluates the body forms with the stream stream "noticed" for input. stream becomes unnoticed afterwards.

The macro is designed to be used with streams created by open-tcp-stream.

**Notes**

1. You do not normally need to use this macro, because all of the standard functions that read from socket streams (read-char and so on) will do this automatically when necessary. However, if you call process-wait yourself with a wait-function that detects new input from a socket stream, then this macro is necessary to cause LispWorks to evaluate the wait-function when there is input on the underlying socket. Without that, there might be a delay before the thread responds to the input.
2. `with-noticed-socket-stream` is not implemented on the Windows platform.

See also `open-tcp-stream`
This chapter describes the LispWorks extensions to symbols in the COMMON-LISP package, which is used by default. This chapter notes only those differences between LispWorks and the ANSI Common Lisp standard.

You should refer to this standard for full documentation about standard Common Lisp symbols. An HTML version, the Common Lisp HyperSpec, is available in the LispWorks IDE via the menu command Help > Manuals > ANSI Common Lisp Standard.

### apropos

**Function**

**Summary**
Searches for interned symbols.

**Package**
common-lisp

**Signature**

```lisp
apropos string &optional package external-only => <no values>
```

**Arguments**

- `string` A string designator.
- `package` A package designator or nil.
- `external-only` A generalized boolean.
The function `apropos` behaves as specified in ANSI Common Lisp. There is an additional optional argument `external-only`, which if true restricts the search to symbols which are external in the searched package or packages. The default value of `external-only` is `nil`.

See also `apropos-list`,
* `describe-print-length`*
* `describe-print-level`*
* `regexp-find-symbols`*

---

### apropos-list

**Function**

**Summary**

Searches for interned symbols.

**Package**

`common-lisp`

**Signature**

`apropos-list string &optional package external-only => symbols`

**Arguments**

- `string` A string designator.
- `package` A package designator or `nil`.
- `external-only` A generalized boolean.

**Values**

- `symbols` A list of symbols, boolean

**Description**

The function `apropos-list` behaves as specified in ANSI Common Lisp. There is an additional optional argument `external-only`, which if true restricts the search to symbols which are external in the searched package or packages. The default value of `external-only` is `nil`.

See also `apropos`
**base-string**

**simple-base-string**

**Summary**
The base string types.

**Package**
common-lisp

**Signature**

- base-string length
- simple-base-string length

**Arguments**

- length
  The length of the string (or *, meaning any).

**Description**

base-string and simple-base-string are the types of base strings and simple base strings respectively.

**See also**

- bmp-string
- text-string

“Character and String types” on page 436

---

**close**

**Generic Function**

**Summary**
The function close is implemented as a generic function.

**Package**
common-lisp

**Signature**

close stream &key abort => result

**Method signatures**

- close :around (stream buffered-stream) &key abort
- close (stream buffered-stream) &key abort

**Arguments**

- stream
  A stream.
- abort
  A generalized boolean.

**Values**

- result
  A boolean.
The standard function `close` is implemented as a generic function. All external resources used by the stream should be freed and true returned when that has been done. The result value for `close` is as per the Common Lisp ANSI specification.

When `stream` is an instance of a subclass of `buffered-stream`, if `abort` is true then any remaining data in the buffer can be discarded. There are two built-in methods on `buffered-stream`. The primary method specialized on `buffered-stream` returns `t`. The other, an around method specialized on `buffered-stream`, checks whether the stream is closed, and if it is does nothing, including not calling the next method, which means not doing any of the primary, before and after methods. If the stream is opened, it flushes the stream buffer if `abort` is `nil`, calls the next method and marks the stream as closed if that method returns true. Thus the only requirement for a primary method specialized on a subclass of `buffered-stream` is that it must close any underlying data source and return true.

**Notes**

1. You should not define an around method on a subclass of `buffered-stream`, as that will happen around the around method on `buffered-stream`. Use before and after methods instead.

2. The `close` method on the `fundamental-stream` class sets a flag for `open-stream-p`.

**See also**

- `buffered-stream`
- `fundamental-stream`
- `open-stream-p`

**Function**

**coerce**

**Summary**

Extends the standard `coerce` function, allowing it to take any Common Lisp type specifier.
Package common-lisp

Signature coerce object result-type => result

Arguments
object A Lisp object.
result-type A type specifier.

Values
result An object of type result-type

Description The function coerce performs those conversions required by the ANSI Common Lisp standard, but a larger set of type specifiers is allowed for coercion.

A type-error is signaled if result cannot be returned as the result-type specifies.

See also concatenate

compile

Function

Summary Compiles a lambda expression into a compiled function.

Package common-lisp

Signature compile name &optional definition => function, warnings-p, failure-p

Arguments
name A function name or nil or a list.
definition A lambda expression or a function.

Description compile calls the compiler to translate a lambda expression into a code vector containing an equivalent sequence of host specific machine code. A compiled function typically runs between 10 and 100 times faster. It is generally worth compiling the most frequently called Lisp functions in a large application during the development phase. The compiler detects a
large number of programming errors, and the resulting code runs sufficiently faster to justify the compilation time, even during development.

Warning messages are printed to *error-output*. Other messages are printed to *standard-output*.

definition and the return values are as specified for Common Lisp. Note that name may be a list not of the form (setf symbol), which is an extension to Common Lisp.

compile also supports a LispWorks-specific extension allowing compile to compile an arbitrary form. When definition is not supplied and name is a list not of the form (setf symbol), compile compiles it as if by compile-file but without any file related processing and does it in-memory, so it has also the same effect as loading. This has a similar effect to compiling a definition in the LispWorks Editor tool, except that there is no source recording. Multiple forms can be compiled in one call by wrapping them with progn. When compile is used this way it always returns nil.

Notes
A compiled function object may be returned. Such compiled function objects are not printable (but see disassemble) other than as #<Function FOO hex-address>.

Compatibility notes
In LispWorks 5.1 and previous versions, warning messages are printed to *standard-output*.

Examples
(defun fn (...) ...) ; interpreted definition for fn
(compile 'fn) ; replace with compiled ; definition

(compile nil '(lambda (x) (* x x))) ; returns compiled squaring function

(compile 'cube '(lambda (x) (* x x x))) ; defun and compile in one
Notes

See declare for a list of the declarations that alter the behavior of the compiler.

See also

compile-file
disassemble
declare

---

**compile-file**

*Function*

**Summary**

Compiles a Lisp source file into a form that both loads and runs faster.

**Package**

common-lisp

**Signature**

`compile-file input-file &key output-file verbose print external-format load => output-truename, warnings-p, failure-p`

**Arguments**

- `input-file` A pathname designator.
- `output-file` A pathname designator, or `:temp`.
- `verbose` A generalized boolean.
- `print` A generalized boolean.
- `external-format` An external format specification.
- `load` A generalized boolean or the keyword `:delete`.

**Values**

- `output-truename` A pathname or `nil`.
- `warnings-p` A generalized boolean.
- `failure-p` A generalized boolean.

**Description**

The function `compile-file` calls the compiler to translate a Lisp source file into a form that both loads and runs faster. A compiled function typically runs more than ten times faster than when interpreted (assuming that it is not spending most
of its work calling already compiled functions). A source file with a .lisp or .lsp extension compiles to produce a file with a .*fasl extension (the actual extension depends on the host machine CPU and the LispWorks implementation). Subsequent use of load loads the compiled version (which is in LispWorks's FASL or Fast Load format) in preference to the source.

In compiling a file the compiler has to both compile each function and top level form in the file, and to produce the appropriate FASL directives so that loading has the desired effect. In particular objects need to have space allocated for them, and top level forms are called as they are loaded.

output-file specifies the location of the output file, relative to the current directory (not the path of the file). If it specifies a directory, then the output file is placed there instead of the same directory as the source. If it contains a file name but not a file type, then the platform specific file type is added and the result specifies the full path of the output file. If output-file has a type, it specifies the full path of the output file. Note that in this case when you want to load the file you will need to add the type to *binary-file-types*. See the example below.

The special value output-file :temp offers a convenient way to specify that the output file is a temporary file in a location that is likely to be writable.

verbose controls the printing of messages describing the file being compiled, the current optimization settings, and other information. If verbose is nil, there are no messages. If verbose is 0, only the "Compiling file..." message is printed. For all other true values of verbose, messages are also printed about:

- compiler optimization settings before the file is processed, and
- multiple matches when input-file does not specify the pathname type, and
any clean down (garbage collection) that occurs during the compilation.

The default value is the value of *compile-verbose*, which defaults to t.

print controls the printing of information about the compilation. It can have the following values. If print is nil, no information is printed. If print is a non-positive number, then only warnings are printed. If print is a positive number no greater than 1, or if print is any non-number object, then the information printed consists of all warning messages and one line of information per function that is compiled. If print is a number greater than 1, then full information is printed. The default value of print is the value of *compile-print*, which has the default value 1.

Warning messages are printed to *error-output*. Other messages are printed to *standard-output*.

external-format is interpreted as for open. The default value is :default.

If load is true, then the file is loaded after compilation. If load is the special value :delete then the compiled file is deleted after loading it. The source file is not affected. This is especially useful when using output-file :temp, to avoid leaving compiled files.

output-truename is the truename of the output file, or nil if that cannot be created.

warnings-p is nil if no conditions of type error or warning were detected during compilation. Otherwise warnings-p is a list containing the conditions.

failure-p is nil if no conditions of type error or warning (other than style-warning) were detected by the compiler, and t otherwise.

Compatibility notes

In LispWorks 5.1 and previous versions, warning messages are printed to *standard-output*. 

647
Examples

```lisp
(compile-file "devel/fred.lisp")
;;; compile fred.lisp to fred.fasl
(compile-file "devel/fred")
;;; does the same thing

(compile-file "test" :load t)
;;; compile test.lisp, then load if successful

(compile-file "program" :output-file "program.abc")
;;; compile "program.lisp" to "program.abc"

(push "abc" sys:*binary-file-types*)
;;; tells LispWorks that files with extension
;;; ".abc" are binaries
```

Notes

See `declare` for a list of the declarations that alter the behavior of the compiler.

The act of compiling a file should have no side effects, other than the creation of symbols and packages as the input file is read by the reader.

By default a form is skipped if an error occurs during compilation. If you need to debug an error during compilation by `compile-file`, set `*compiler-break-on-error*`.

During compilation of a file `foo.lisp` (on an Intel Macintosh, for example) a temporary output file named `t_foo.xfasl` is used, so that an unsuccessful compile does not overwrite an existing `foo.xfasl`.

LispWorks uses the following naming conventions for fasl files, and it is recommended that you should use them too, to ensure correct operation of `load` and so on.

<table>
<thead>
<tr>
<th>Machine/Implementation</th>
<th>Fasl Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 Windows/32-bit LispWorks</td>
<td>.ofasl</td>
</tr>
<tr>
<td>x64 Windows/64-bit LispWorks</td>
<td>.64ofasl</td>
</tr>
</tbody>
</table>
You can find the fasl file extension appropriate for your machine by looking at the variable `*binary-file-type*`. The variable `*binary-file-types*` contains a list of all the file extensions currently recognized by `load`, `require` and `load-data-file` (in addition to `*binary-file-type*`).

**Compatibility notes**

1. In LispWorks for Windows 4.4 and previous versions, the fasl file extension is `.fsl`. This changed in LispWorks 5.0.

<table>
<thead>
<tr>
<th>Machine/Implementation</th>
<th>Fasl Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 Linux/32-bit LispWorks</td>
<td>.ufasl</td>
</tr>
<tr>
<td>amd64 Linux/64-bit LispWorks</td>
<td>.64ufasl</td>
</tr>
<tr>
<td>ARM Linux/32-bit LispWorks</td>
<td>.rfasl</td>
</tr>
<tr>
<td>ARM Linux/64-bit LispWorks</td>
<td>.64rfasl</td>
</tr>
<tr>
<td>x86 FreeBSD/32-bit LispWorks</td>
<td>.ffasl</td>
</tr>
<tr>
<td>amd64 FreeBSD/64-bit LispWorks</td>
<td>.64ffasl</td>
</tr>
<tr>
<td>AIX PowerPC/32-bit LispWorks</td>
<td>.ifas1</td>
</tr>
<tr>
<td>AIX PowerPC/64-bit LispWorks</td>
<td>.64ifas1</td>
</tr>
<tr>
<td>SPARC/32-bit LispWorks</td>
<td>.wfasl</td>
</tr>
<tr>
<td>SPARC/64-bit LispWorks</td>
<td>.64wfasl</td>
</tr>
<tr>
<td>x86 Solaris/32-bit LispWorks</td>
<td>.sfas1</td>
</tr>
<tr>
<td>amd64 Solaris/64-bit LispWorks</td>
<td>.64sfas1</td>
</tr>
<tr>
<td>Intel Macintosh/32-bit LispWorks</td>
<td>.xfas1</td>
</tr>
<tr>
<td>Intel Macintosh/64-bit LispWorks</td>
<td>.64xfas1</td>
</tr>
<tr>
<td>LispWorks for iOS Runtime simulator</td>
<td>.xcfasl</td>
</tr>
<tr>
<td>LispWorks for iOS Runtime</td>
<td>.rfas1</td>
</tr>
<tr>
<td>LispWorks for Android Runtime</td>
<td>.rfas1</td>
</tr>
</tbody>
</table>
2. In LispWorks for Linux 4.4 and previous versions, the fasl file extension is `.ufsl`. This changed in LispWorks 5.0.

See also

- `compile`
- `compile-file-if-needed`
- `*compiler-break-on-error*`
- `disassemble`

**concatenate**

**Function**

**Summary**

Extends the standard `concatenate` function allowing it to take any Common Lisp type.

**Package**

`common-lisp`

**Signature**

`concatenate result-type &rest sequences => result-sequence`

**Arguments**

- `result-type` A type specifier.
- `sequences` A sequence.

**Values**

- `result-sequence` A sequence.

**Description**

The function `concatenate` has been extended to take any Common Lisp type. The `result-sequence` will be of type `result-type` unless this is not possible, in which case a `type-error` is signaled.

See also

- `coerce`

**declaim**

**Macro**

**Summary**

Established a specified declarations.

**Package**

`common-lisp`
Signature  
\texttt{declaim \&rest declarations}

Arguments  
\texttt{declarations}  
Declaration forms.

Description  
The macro \texttt{declaim} behaves as specified in the ANSI Common Lisp Standard with one exception: for a top-level call to \texttt{declaim}, optimize declarations are omitted from the compiled binary file. This is useful because you are unlikely to want to change these settings outside of that file.

See also  
\texttt{compile-file}  
\texttt{declare}  
\texttt{proclaim}

declare  
\textit{Special Form}

Summary  
Declares a variable as special, provides advice to the Common Lisp system, or helps the programmer to optimize code.

Package  
\texttt{common-lisp}

Signature  
\texttt{declare declaration*}

Arguments  
\texttt{declaration}  
A declaration specifier, not evaluated.

Values  
The special form \texttt{declare} behaves computationally as if it is not present (other than to affect the semantics), and is only allowed in certain contexts, such as after the variable list in a \texttt{let}, \texttt{do}, \texttt{defun} and so on.

(Consult the syntax definition of each special form to see if it takes \texttt{declare} forms and/or documentation strings.)

Description  
There are three distinct uses of \texttt{declare}: one is to declare Lisp variables as “special” (this affects the semantics of the appropriate bindings of the variables), the second is to pro-
vide advice to help the Common Lisp system (in reality the compiler) run your Lisp code faster or with more sophisticated debugging options, and the third (using the :explain declaration) is to help you optimize your code.

If you use declare to specify types (and so eliminate type-checking for the specified symbols) and then supply the wrong type, you may obtain a “Segmentation Violation”. You can check this by interpreting the code (rather than compiling it).

The declare special form can be used as documented in the Common Lisp HyperSpec as well as with the following extensions:

- **hcl:special-global** declares that the symbol is never bound.
  
  In SMP LispWorks the compiler signals error if it detects that a symbol declared as hcl:special-global is bound, and at run time it also signals an error.
  
  In non-SMP LispWorks the compiler gives an error, but there is no run time check. The run time behavior is the same as cl:special, with all accesses to the symbol in low safety.
  
  hcl:special-global is very useful, and because of the checks it is reasonably safe. It is useful not only for speed, but also to guard against unintentionally binding variables that should not be bound.

  See also defglobal-parameter.

- **hcl:special-dynamic** declares that the symbol is never accessed outside the dynamic scope of the binding.
  
  In high safety code accessing the symbol outside the scope of binding signals an error. In low safety code it may result in unpredictable behavior.
  
  In non-SMP LispWorks the only effect of this declaration is to make all access to the variable low safety.
hcl:special-dynamic is useful, but because it can lead to unpredictable behavior you need to ensure that you test your program in high safety when you use it.

• hcl:special-fast-access declares that a symbol should be “fast access”.

The semantics of the declaration is the same as cl:special, except that access to the variable is low safety. In addition, the compiler compiles access to the symbol in a way that speeds up the access, but also introduces a tiny reduction in the speed of the whole system. The balance between these effects is not obvious.

It is not obvious where hcl:special-fast-access is useful. If you can ensure that the symbol is always bound or never bound then hcl:special-dynamic or hcl:special-global are certainly better.

• hcl:lambda-list specifies the value to be returned when a programmer asks for the arglist of a function.

• values specifies the value to be returned when you ask for a description of the results of a function.

• hcl:invisible-frame specifies that calls to this function should not appear in a debugger backtrace.

• hcl:alias specifies that calls to this function should be displayed as calls to an alternative function in a debugger backtrace.

• hcl:lambda-name declares the name of the surrounding lambda.

• :explain controls messages printed by the compiler while it is processing forms.

You can also use define-declaration to add your own declarations, which do not affect compilation but are useful for code walkers.
This section documents the `hcl:lambda-name` declaration, which declares the name of the surrounding lambda. This declaration is useful only for a lambda that becomes a standalone function, that is lambda forms that are passed to `function`.

The dspec of the function that is returned by `function` is specified by the second element in the declaration. In the special case when the second element is a two-element list starting with the symbol `subfunction`, the dspec is that list with the dspec of the parent function added as a third element. For example, if you have:

```lisp
(defun my-parent (x)
  #'(lambda (y)
      (declare (lambda-name (subfunction sub-name)))
      (* x y)))
```

then the dspec of the subfunction that `my-parent` returns would be `(subfunction sub-name my-parent)`.

`hcl:lambda-name` is useful for debugging purposes and does not affect the behavior of the program. There are two different situations when `hcl:lambda-name` is useful:

- In a defining form that has a similar effect to the effect of `defun` (that is creating a "top-level" function at load-time). In this case, you should also use `def` to be able to locate the source. For example, look at the output of

  ```lisp
  (pprint (macroexpand-1 '(defun func-name ())))
  ```

- In a "run time" subfunction (that is a subfunction created by a code at run time by executing `(function (lambda ..))`). In this case, you should be using the `(subfunction sub-name)` form above, so the recorded name contains the dspec of the parent function, otherwise the debugger will not be able to find the source from the subfunction.
hcl:lambda-name will also modify the function name of flet and labels, but these already have a name, so this is not often useful.

Naming functions and subfunctions is useful because it makes it easier to understand the flow of control when you see them in a backtrace. For subfunctions, it makes it easier to trace and advise them (see trace and defadvice).

Description of :explain

The remainder of this description documents the syntax and use of :explain declarations.

```
declaration ::= (:explain option*)
```

```
option ::= optionkey | (optionkey optionvalue)
```

```
```

The :explain declaration controls messages printed by the compiler while it is processing forms. The declaration can be used with proclaim or declaim as a top level form to give it global or file scope. It can also be used at the start of a #'lambda form or function body to give it the scope of that function. The declaration has unspecified effect when used in other contexts, for example in the body of a let form.

An :explain declaration consists of a set of options of the form (optionkey optionvalue) which associates optionvalue with optionkey or optionkey which associates t with optionkey. By default, all of the optionkeys have an associated value nil. All optionkeys not specified by a declaration remain unchanged (except for the special action of the :none optionkey described below).

The optionkey should be one of the following:

```
:none Set value associated with all optionkeys to nil. This turns off all explanations.
```
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:variables</td>
<td>If <code>optionvalue</code> is non-nil, list all the variables of each function, specifying whether they are floating point or not.</td>
</tr>
<tr>
<td>:types</td>
<td>If <code>optionvalue</code> is non-nil, print information about compiler transformations that depend on declared or deduced type information.</td>
</tr>
<tr>
<td>:floats</td>
<td>If <code>optionvalue</code> is non-nil, print information about calls to functions that may allocate floats.</td>
</tr>
<tr>
<td>:non-floats</td>
<td>If <code>optionvalue</code> is non-nil, print information about calls to functions that may allocate non-float numbers, for example bignums.</td>
</tr>
<tr>
<td>:all-calls</td>
<td>If <code>optionvalue</code> is non-nil, print information about calls to normal functions.</td>
</tr>
<tr>
<td>:all-calls-with-arg-types</td>
<td>If <code>optionvalue</code> is non-nil, print the argument types for calls to normal functions. Must be combined with :all-calls.</td>
</tr>
<tr>
<td>:calls</td>
<td>A synonym for :all-calls.</td>
</tr>
<tr>
<td>:boxing</td>
<td>If <code>optionvalue</code> is non-nil, print information about calls to functions that may allocate numbers, for example floats or bignums.</td>
</tr>
<tr>
<td>:print-original-form</td>
<td>If <code>optionvalue</code> is non-nil, modifies the :all-calls, :floats and :non-floats explanations to include the original source code form that contains the call.</td>
</tr>
<tr>
<td>:print-expanded-form</td>
<td>If <code>optionvalue</code> is non-nil, modifies the :all-calls, :floats and :non-floats explanations to include the macroexpanded source code form that contains the call.</td>
</tr>
</tbody>
</table>
:print-length* for :all-calls, :floats and :non-floats explanations.

:print-level* for :all-calls, :floats and :non-floats explanations.

Example

```
(defun foo (arg)
  (declare
    (:explain :variables)
    (optimize (float 0)))
  (let* ((double-arg (coerce arg 'double-float))
         (next (+ double-arg 1d0))
         (other (* double-arg 1/2)))
    (values next other)))
```

;;- Variables with non-floating point types:
;;- ARG OTHER
;;- Variables with floating point types:
;;- DOUBLE-ARG NEXT

See also

“Compiler control” on page 104
compile
compile-file
proclaim
define-declaration
declaration-information

defclass

Macro

Summary
Remains as defined in ANSI Common Lisp, but extra control
over parsing of class options and slot options, optimization
of slot access, and checking of initargs, is provided.

Package
common-lisp

Description
The macro defclass is as defined in the ANSI standard with
the following extensions.
For extra class options, you may need to define the way these are parsed at `defclass` macroexpansion time. See `process-a-class-option` for details.

For non-standard slot options, you may need to define the way these are parsed at `defclass` macroexpansion time. See `process-a-slot-option` for details.

By default, standard slot accessors are optimized such that they do not call `slot-value-using-class`. This optimization can be switched off using the `:optimize-slot-access nil` class option.

To add valid initialization arguments for the class, use the class option `:extra-initargs`. The argument passed via this option is evaluated, and should return a list of extra initialization arguments for the class. `make-instance` and other CLOS initializations (see `set-clos-initarg-checking`) will treat these as valid when checking their arguments.

**Compatibility notes**

1. When a class is redefined, its extra initargs are always reset.
2. In early versions of LispWorks 4.3, extra initargs were not reset when a class was redefined without specifying extra initargs.

**Example**

This session illustrates the effects of the `:optimize-slot-access` class option. When true, slot access is more efficient but note that `slot-value-using-class` is not called.
CL-USER 26 > (compile '(defclass foo ()
    ((a :type fixnum
      :initarg :a
      :reader foo-a))))
NIL

CL-USER 27 > (compile '(defclass bar ()
    ((a :type fixnum
      :initarg :a
      :reader bar-a))
    (:optimize-slot-access nil)))
NIL

CL-USER 28 > (setf *foo* (make-instance 'foo :a 42)
                *bar* (make-instance 'bar :a 99))
#<BAR 21D33D4C>

CL-USER 29 > (progn
    (time (dotimes (i 1000000)
           (foo-a *foo*))
    (time (dotimes (i 1000000)
           (bar-a *bar*))))
Timing the evaluation of (DOTIMES (I 1000000) (FOO-A *FOO*))

user time    =      0.328
system time  =      0.015
Elapsed time =   0:00:00
Allocation   = 2280 bytes standard / 11002882 bytes conses
0 Page faults

Timing the evaluation of (DOTIMES (I 1000000) (BAR-A *BAR*))

user time    =      0.406
system time  =      0.015
Elapsed time =   0:00:00
Allocation   = 4304 bytes standard / 11004521 bytes conses
0 Page faults

NIL

CL-USER 30 > (trace
    (clos:slot-value-using-class
     :when
    (and (member (first *traced-arglist*)

659
This session illustrates the :extra-initargs class option:
CL-USER 46 > (defclass a () ()
    (:extra-initargs '(:a-initarg)))
#<STANDARD-CLASS A 21C2E4FC>

CL-USER 47 > (defclass b (a) ()
    (:extra-initargs '(:b-initarg)))
#<STANDARD-CLASS B 2068573C>

CL-USER 48 > (defclass c (a) ())
#<STANDARD-CLASS C 22829D44>

CL-USER 49 > (make-instance 'b :a-initarg "A" :b-initarg "B")
#<B 2068BCE4>

CL-USER 50 > (make-instance 'c :a-initarg "A" :b-initarg "B")

Error: MAKE-INSTANCE is called with unknown keyword :B-INITARG among the arguments (C :A-INITARG "A" :B-INITARG "B") which is not one of (:A-INITARG).

1 (continue) Ignore the keyword :B-INITARG
2 (abort) Return to level 0.
3 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :? for other options

CL-USER 51 : 1 >

See also
process-a-class-option
process-a-slot-option

defpackage

Macro

Summary
Remains as defined in Common Lisp, but see *handle-existing-defpackage* for an extension.

Package
common-lisp

Signature
defpackage defined-package-name [[option]] => package
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>defined-package-name</td>
<td>A string designator.</td>
</tr>
<tr>
<td>option</td>
<td>Keyword options.</td>
</tr>
<tr>
<td>add-use-defaults</td>
<td>A keyword</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>package</td>
<td>A package.</td>
</tr>
</tbody>
</table>

Description

The macro `defpackage` is as defined in the ANSI standard, with the inclusion of the `:add-use-defaults` keyword. However, the standard explicitly declines to define what `defpackage` does if a package named `defined-package-name` already exists and is in a state that differs from that described by the `defpackage` form.

Therefore an extension has been written that allows you to select between alternative behaviors. See `*handle-existing-defpackage*` for full details.

One non-standard `option` is supported. `:add-use-defaults`, with a true argument, causes the package `defined-package-name` to inherit from the following packages (as well as any explicitly specified by the `:use` option):

- common-lisp
- lispworks
- harlequin-common-lisp

Example

```lisp
(defpackage "MY-PACKAGE" (:use "CAPI")
 (:add-use-defaults t))
```

```lisp
(package-use-list "MY-PACKAGE")
=>
(#<PACKAGE COMMON-LISP> #<PACKAGE LISPWORKS> #<PACKAGE HARLEQUIN-COMMON-LISP> #<PACKAGE CAPI>)
```

See also

`*handle-existing-defpackage*`
describe

Function

Summary
Remains as defined in ANSI Common Lisp. Additionally, you can control the depth at which slots inside arrays, structures and conses are described.

Package common-lisp

Signature
describe object &optional stream => <no-values>

Arguments
object An object.
stream An output stream designator.

Description
The function describe displays information about the object object to the stream indicated by stream, as specified in ANSI Common Lisp.

Arrays, structures and conses are described recursively up to the depth given in the value of the variable *describe-level*. Beyond that depth, objects are simply printed.

See also
*describe-length*
*describe-level*
*describe-print-length*
*describe-print-level*

directory

Function

Summary
Determines which files on the system have names matching a given pathname.

Package common-lisp

Signature
directory pathname &key test directories flat-file-namestring link-transparency non-existent-link-destinations => pathnames

Arguments
pathname A pathname, string, or file-stream.
test Filtering test (only pathnames matching the test are collected).

directories A boolean controlling whether non-matching directories are included in the result.

flat-file-namestring A generalized boolean.

link-transparency If nil, then symbolic links are not followed. This means that returned names are not necessarily truenames, but has the useful feature that the pathname-directory of each pathname returned is the directory supplied as argument.

The default value of link-transparency is given by the special variable, *directory-link-transparency*, which has initial value t on non-Windows platforms. By setting this variable to nil, you can get the old behavior of directory. On Windows, where the file system does not normally support symbolic links, this variable is initially nil.

non-existent-link-destinations

If this is non-nil, then the pathname pointed to by a symbolic link appears in the output whether or not this file actually exists. If :link-transparency is non-nil and :non-existent-link-destinations is nil (this is the default on non-Windows platforms), then symbolic links to nonexistent files do not appear.

The default value is nil.

Values pathnames A list of physical pathnames.
**Description**

`directory` collects all the pathnames matching the given pathname.

`directory` returns truenames, conforming to the ANSI specification for Common Lisp. Some programs may depend on the old behavior, however (and `directory` is slower if it has to find the truename for every file in the directory), and so two keyword arguments are available so that the old behavior can still be used: `link-transparency` and `non-existent-link-destinations`.

Because truenames are now returned, the entries `. and `.. no longer show up in the output of `directory`. This means, for instance, that

```lisp
(directory #P"/usr/users/"
```

does not include #P"/usr", which is the truename of #P"/usr/users/.."

The specification is unclear as to the appropriate behavior of `directory` in the presence of links to non-existent files or directories. For example, if the directory contains `foo`, which is a symbolic link to `bar`, and there is no file named `bar`, should `bar` show up in the directory listing? A keyword argument has been added which lets you control this behavior.

`directory` returns a single pathname if called with a non-wild (fully-specified) `pathname`. LispWorks truenames are fully-specified, so this affects recursive calls to `directory`.

directories, if non-nil, causes paths of directories that are subdirectories of the directory of the argument `pathname` to be included in the result, even if they do not match `pathname` in the name, type or version components. The default value of `directories` is `nil`.

When `flat-file-namestring` is non-nil, `directory` matches the `file-namestring` of `pathname` as a flat string, rather than a
The COMMON-LISP Package

pathname name and pathname type. The default value of *flat-file-namestring* is `nil`.

Notes

1. The Search files tool in the LispWorks IDE uses this option when the *Match flat file-namestring* option is selected. See the *LispWorks IDE User Guide* for more information about the Search Files tool.

2. File names containing the character * cannot be handled by LispWorks. This is because LispWorks uses * as a wildcard, so there can be confusion if a file name containing * is created, for example in the *pathnames* returned by *directory*.

3. The function *fast-directory-files* can be used for faster operations when operating on directories with large number of files.

Compatibility notes

The :check-for-subs argument, implemented in LispWorks 4.0.1 and previous versions, has been removed. This argument controlled whether directories in the result have null name components. This option is no longer valid since ANSI Common Lisp specifies that *directory* returns true-names.

Example

```lisp
CL-USER 1 > (pprint (directory "."))

(#P"C:/Program Files/LispWorks/readme-6-1.txt"
 #P"C:/Program Files/LispWorks/lispworks-6-1-0-x86-win32.exe"
 #P"C:/Program Files/LispWorks/license-6-1.txt"
 #P"C:/Program Files/LispWorks/lib/*")

This session illustrates the effect of the *directories* argument:
CL-USER 5 > (pprint (directory "/tmp/t*"))

(#P"/tmp/test.lisp" #P"/tmp/test2/" #P"/tmp/test1/"

CL-USER 6 > (pprint (directory "/tmp/t*" :directories t))

(#P"/tmp/patches/"
 #P"/tmp/test.lisp"
 #P"/tmp/test2/"
 #P"/tmp/opengl/"
 #P"/tmp/test1/"
 #P"/tmp/mnt/"

This example illustrates directory returning a single path-name in its result when given a full-specified pathname:

CL-USER 1 > (directory
    (make-pathname :host "H"
        :device :unspecific
        :directory (list :absolute "tmp")
        :name :unspecific
        :type :unspecific
        :version :unspecific))

(#P"H:/tmp/"

The next two examples illustrate the effect of flat-file-namestring. Suppose the directory dir contains files interp.exe and file.lisp.

This call matches interp.exe, where the name interp ends with p, but does not match file.lisp, where the name file ends with e:

(directory "dir/*p")

The next call matches file.lisp, where the namestring file.lisp ends with p, but does not match interp.exe, where the namestring interp.exe ends with e:

(directory "dir/*p" :flat-file-namestring t)

See also fast-directory-files
truename
**disassemble**

*Function*

**Summary**
Prints the machine code of a compiled function.

**Package**
common-lisp

**Signature**
disassemble name-or-function => nil

**Arguments**
name-or-function Either a function object, a lambda expression or a symbol with a function definition.

**Description**
This function prints the machine code of a compiled function, to *standard-output*.

If the function denoted by name-or-function is not compiled then it is first compiled using the function compile. This happens if name-or-function is a lambda expression or an symbol naming an interpreted function.

An error is signaled if name-or-function is not suitable.

**Examples**

```lisp
(disassemble #'(lambda (x) (progn x)))
(disassemble 'cons)
(disassemble #'map)
```

**Notes**
The output from disassemble lacks useful information such as local and lexical variable names. The representation of integers or characters or Lisp objects in general is not easily readable without detailed knowledge of the internals of the Lisp system and the host machine instruction set.

**See also**
compile
compile-file

---

**documentation**

*Generic Function*

**Summary**
Returns the documentation string if available.
The generic function documentation operates as specified in the ANSI Common Lisp standard. Additional methods with signatures:

\[
\text{documentation (dspec t) (doc-type (eql 'dspec:dspec))}
\]

\[
\text{(setf documentation) new-value (dspec t) (doc-type (eql 'dspec:dspec))}
\]

are provided.

This method allows finding or setting the documentation string when you know the dspec. See Chapter 7, “Dspecs: Tools for Handling Definitions” for information about dspecs.

dspec must be a dspec, but it can be non-canonical. This method canonicalizes dspec and then calls documentation with the name as the first argument and the appropriate dspec class name as the second, thereby calling a standard documentation method.

If you define your own type of definitions (def-saved-value for example) with define-dspec-class you can add methods on documentation for your dspec class:

\[
\text{(documentation (dspec t) (doc-type (eql 'def-saved-value))}
\]

This allows commands in the LispWorks IDE such as Expression > Documentation to display the documentation.

double-float

A subtype of float.

Type
**Package**
- common-lisp

**Signature**
- double-float

**Description**
- double-float is disjoint from short-float and single-float in all LispWorks implementations in version 5.0 and later.

**Compatibility notes**
- In LispWorks 4.4 and previous on Windows and Linux platforms, all floats are of type double-float. However, there are distinct specialized array types (array single-float), with single precision, and (array double-float), with double precision.

**See also**
- long-float
- parse-float
- short-float
- single-float

---

### *features*

**Summary**
The features list.

**Package**
- common-lisp

**Initial value**
- A list containing :lispworks. The actual value varies depending on the platform.

**Description**
The following features can be used to distinguish between platforms, or characteristics of the platform or of the LispWorks implementation.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:solaris2</td>
<td>Solaris2</td>
</tr>
<tr>
<td>:aix</td>
<td>AIX PowerPC</td>
</tr>
<tr>
<td>:svr4</td>
<td>System 5 Release 4 machine (for example Solaris2)</td>
</tr>
</tbody>
</table>
:linux   Linux
:darwin   The variant of FreeBSD underlying Mac OS X.
:unix     Unix, including all of the above.
:mswindows Microsoft Windows, including 32-bit and 64-bit.
:lispworks-64bit  64-bit LispWorks.
:x86      All images that run on the x86 architecture have this feature. This includes Intel Macintosh, FreeBSD, x86/x86_x64 Linux (32-bit), x86/x64 Solaris (32-bit) and Windows (32-bit).

**Note:** 64-bit LispWorks does not have this feature.

:amd64, :x86-64, :x64  Images that run on the amd64/x86_64/x64 architecture have each of these features. This includes x86_x64 Linux (64-bit), FreeBSD (64-bit), x86/x64 Solaris (64-bit) and Windows (64-bit).

:arm      Images that run on 32-bit ARM architecture.
:arm64    Images that run on 64-bit ARM architecture.
:sparc    Images that run on SPARC architecture.
:powerpc  Images that run on PowerPC architecture.
:android-delivery Images generating Android runtimes.
:ios-delivery Images generating iOS runtimes.
The compiler targets a little-endian machine, for instance x86.

The following features are present in LispWorks with the meanings defined for ANSI CL:

:ansi-cl
:common-lisp
:ieee-floating-point

**Conditionalization for the LispWorks implementations**

Code can distinguish the seventeen current LispWorks implementations like this:
#+(and :mswindows :x86)
"LispWorks (32-bit) for Windows"
#+(and :mswindows :x86-64)
"LispWorks (64-bit) for Windows"
#+(and :linux :x86)
"LispWorks (32-bit) for x86/x86_64 Linux"
#+(and :linux :x86-64)
"LispWorks (64-bit) for x86_64 Linux"
#+(and :linux :arm
  (not :android-delivery)
  (not :ios-delivery))
"LispWorks (32-bit) for ARM Linux"
#+(and :linux :arm64
  (not :android-delivery)
  (not :ios-delivery))
"LispWorks (64-bit) for ARM Linux"
#+(and :freebsd :x86)
"LispWorks (32-bit) for FreeBSD"
#+(and :freebsd :x86-64)
"LispWorks (64-bit) for FreeBSD"
#+(and :aix :lispworks-32bit)
"LispWorks (32-bit) for AIX"
#+(and :aix :lispworks-64bit)
"LispWorks (64-bit) for AIX"
#+(and :darwin :x86 (not :ios-delivery))
"LispWorks (32-bit) for Macintosh"
#+(and :darwin :x86-64)
"LispWorks (64-bit) for Macintosh"
#+(and :solaris2 :x86)
"LispWorks (32-bit) for x86/x64 Solaris"
#+(and :solaris2 :x86-64)
"LispWorks (64-bit) for x86/x64 Solaris"
#+(and :sparc :lispworks-32bit)
"LispWorks (32-bit) for SPARC Solaris"
#+(and :sparc :lispworks-64bit)
"LispWorks (64-bit) for SPARC Solaris"
#+:android-delivery
"LispWorks (32-bit) for Android Runtime"
#+(and :ios-delivery :x86)
"LispWorks (32-bit) for iOS Runtime simulator"
#+(and :ios-delivery :x86-64)
"LispWorks (64-bit) for iOS Runtime simulator"
#+(and :ios-delivery :arm)
"LispWorks (32-bit) for iOS Runtime"
#+(and :ios-delivery :arm)
"LispWorks (64-bit) for iOS Runtime"
Conditionalization for LispWorks versions

The following features can be used to distinguish between versions of LispWorks:

:lipsworks4       All major version 4 releases.
:lipsworks4.4     Release 4.4.x
:lipsworks5       All major version 5 releases.
:lipsworks5.0     Release 5.0.x
:lipsworks5.1     Release 5.1.x
:lipsworks6       All major version 6 releases.
:lipsworks6.0     Release 6.0.x
:lipsworks6.1     Release 6.1.x
:lipsworks7       All major version 7 releases.
:lipsworks7.0     Release 7.0.x
:lipsworks7.1     Release 7.1.x

Code using new LispWorks functionality should be conditionalyzed only using features representing earlier versions, so as to future-proof your code:

(defun *feature-added-in-LispWorks-7.1*
  (+ (or lipsworks4 lipsworks5 lipsworks6 lipsworks7.0) nil
      (or lipsworks4 lipsworks5 lipsworks6 lipsworks7.0) t)
)

This is because a feature added in LispWorks 7.1 will generally also be in LispWorks 7.2, LispWorks 8.0 and all later versions.

Similarly:

(defun *feature-added-in-LispWorks-7.0*
  (+ (or lipsworks4 lipsworks5 lipsworks6) nil
      (or lipsworks4 lipsworks5 lipsworks6) t)
)

or:
(defvar *feature-added-in-LispWorks-6.1*  
  #+(or lispworks4 lispworks5 lispworks6.0) nil  
  #-(or lispworks4 lispworks5 lispworks6.0) t)

We have seen several problematic examples like this:

(defvar *feature-added-in-LispWorks-6.0*  
  #+lispworks6 t  
  #-lispworks6 nil)

which breaks in LispWorks 7.0, because that release does not contain the :lispworks6 feature.

In general you should use use the :lispworksx and :lispworksx.y features "in reverse". That is, make your code work for the latest version of LispWorks and then add conditionalization for any previous versions that you want to support, if needed.

**Conditionalization for the LispWorks architectures**

Every LispWorks 5, LispWorks 6 and LispWorks 7 image has exactly one of the features :lispworks-32bit and :lispworks-64bit.

The two LispWorks architectures, 32-bit and 64-bit, can be distinguished by the features: lispworks-32bit or :lispworks-64bit.

**Notes**

1. For a LispWorks image with the CAPI loaded, :capi will appear on *features*.

2. LispWorks for Macintosh supports the native Mac OS X Cocoa-based GUI and the X11/GTK+ GUI. If you need to test for which of these libraries is loaded, check for the features :cocoa and :gtk. The X11/Motif GUI is also available by evaluating (require "capi-motif") in the GTK+ image.
3. Sometimes it is necessary to write code that examines *features* at load-time or run-time. For example this is true when you put platform-dependent code in fasl files that are shared between multiple platforms.

**input-stream-p**

*Generic Function*

**Summary**
A generic function that determines if an object is an input stream.

**Package**
common-lisp

**Signature**
input-stream-p stream => result

**Arguments**
stream A stream.

**Values**
result A generalized boolean.

**Description**
The predicate input-stream-p is implemented as a generic function. The default method returns t if stream is an input stream. If the user wants to implement a stream with no inherent directionality (and thus does not include fundamental-input-stream or fundamental-output-stream) but for which the directionality depends on the instance, then a method should be provided for input-stream-p.

**Examples**
There is an example in “Stream directionality” on page 400.

**See also**
fundamental-input-stream
output-stream-p

**interactive-stream-p**

*Function*

**Summary**
A generic function that determines if an object is an interactive stream.
### interactive-stream-p

**Package**: cl  
**Signature**: `interactive-stream-p stream => result`  
**Arguments**: `stream` A stream.  
**Values**: `result` A generalized boolean.  
**Description**: The predicate `interactive-stream-p` is implemented as a generic function. The `fundamental-stream` class provides a default method that returns `nil`.  
**See also**: `input-stream-p`, `output-stream-p`

### load-logical-pathname-translations

**Function**  
**Summary**: Searches for and loads the definition of a logical host, if not already defined.  
**Package**: cl  
**Signature**: `load-logical-pathname-translations host => just-loaded`  
**Arguments**: `host` A logical host, expressed as a string.  
**Values**: `just-loaded` A generalized boolean  
**Description**: This function loads the translations for `host` by loading the file `host.lisp` from the LispWorks directory `translations`.  
**Example**: `(load-logical-pathname-translations "EDITOR-SRC")`
### long-float

**Summary**
A subtype of `float`.

**Package**
common-lisp

**Signature**
long-float

**Description**
`long-float` is the same type as `double-float` in LispWorks, on all platforms.

**See also**
double-float
parse-float
short-float
single-float

### long-site-name

**Summary**
Identifies the physical location of the computer.

**Package**
common-lisp

**Signature**
long-site-name => description
(setf long-site-name) description => description

**Arguments**
description A string or nil.

**Description**
The function `long-site-name` returns a string identifying the physical location of the computer. This should be set using `(setf long-site-name)` when you configure your LispWorks image.

**See also**
short-site-name
**loop**

**Macro**

**Summary**
A macro that performs iteration.

**Package**
cl

**Signature**

```
loop {for|as} var  [type-spec]
    being {the|each}{records|record}
    {in|of} query-expression => result
```

**Arguments**

- `var` A variable.
- `query-expression` A SQL query-statement

**Values**

- `result` An object.

**Description**
The Common Lisp `loop` macro has been extended with a clause for iterating over query results. This extension is available only when the SQL interface has been loaded. See Chapter 47, “The SQL Package”. For a full description of the rest of the Common Lisp `loop` facility, see the Common Lisp Hyperspec.

Each iteration of the loop assigns the next record of the table to the variable `var`. The record is represented in Lisp as a list. Destructuring can be used in `var` to bind variables to specific attributes of the records resulting from `query-expression`. In conjunction with the panoply of existing clauses available from the loop macro, the new iteration clause provides an integrated report generation facility.

**Example**

This extended `loop` example, on each record returned as a result of the query, binds `name`, finds the salary (if any) from an associated hash-table, and for salaries greater than 20000: increments a count, accumulates the salary, and prints the details. Finally, the average salary is printed.
(loop
   for (name) being each record in
   [select [ename] :from [emp]]
   as salary = (gethash name *salary-table*)
   initially (format t "%-6-20A-10D" 'name 'salary)
   when (and salary (> salary 20000))
   count salary into salaries
   and sum salary into total
   and do (format t "%-6-20A-10D" name salary)
   else
   do (format t "%-6-20A-10A" name "N/A")
   finally
   (format t "%-2&Av Salary: -10D" (/ total salaries)))

See also
   do-query
   map-query
   query
   select

make-array

Function

Summary
Creates and returns a new array which, in addition to the standard functionality, can be a weak array or statically allocated.

Package
common-lisp

Signature
make-array dimensions &key element-type initial-element initial-contents adjustable fill-pointer displaced-to displaced-index-offset weak allocation single-thread => new-array

Arguments

weak A generalized boolean.

allocation nil or one of the keywords :new, :static, :old and :long-lived.

single-thread A generalized boolean.
The standard definition of `make-array` is extended to accept the keyword arguments `:weak`, `:allocation` and `:single-thread`.

If `weak` is non-nil, then `displaced-to` must be `nil` and if `element-type` is supplied it must have `upgraded-array-element-type t`, otherwise an error is signaled. That is, you cannot make a weak array which is displaced or has `array-element-type` other than `t`. When `weak` is non-nil, it makes `new-array` weak.

If `weak` is `nil`, then `make-array` behaves in the standard way, and `new-array` is not weak. The value `weak` defaults to `nil`. In 64-bit LispWorks, `allocation` cannot be used with `weak` and the length of a weak array must be less than `4194304 (222)` elements.

See `set-array-weak` for a description of weak arrays.

The possible values for `allocation` have the following meanings:

- `:new` Allocate the array normally.
- `nil` Same meaning as `:new`. This is the default value.
- `:static` Allocate the array in a static segment.
- `:long-lived` Allocate the array assuming it is going to be long-lived.
- `:old` Same meaning as `:long-lived`

Arrays (including strings) that are passed by address to foreign functions must be static, and so must should be created with `:allocation :static`.

Allocation with `:old` or `:long-lived` is useful when you know that the array will be long-lived, because your program will avoid the overhead of promoting it to the older generations.
If `single-thread` is true then the system knows that `new-array` will always be accessed in a single thread context. That makes some operations faster, in particular `vector-pop` and `vector-push`. The default value of `single-thread` is `nil`.

Compatibility notes

`allocation` can also be a fixnum `n` but this is now deprecated. The intent was to allocate the array in generation `n`, however the allocation is not actually guaranteed to be in the specified generation (although it will be in almost every call).

See also

- `array-weak-p`
- `set-array-single-thread-p`
- `set-array-weak`
- “Freeing of objects by the GC” on page 165

**make-hash-table**

*Function*

**Summary**

Creates and returns a new hash table which, in addition to the standard functionality, can have a user-defined test, a user-defined hash function, and be a weak hash table.

**Package**

`common-lisp`

**Signature**

```
make-hash-table &key test size rehash-size rehash-threshold hash-function weak-kind single-thread free-function => hash-table
```

**Arguments**

- `test` A designator for a function of two arguments, which returns `t` if they should be regarded as the same and `nil` otherwise.
- `hash-function` A designator for a function of one argument, which returns a hash value.
- `weak-kind` `t`, `nil`, or one of the keywords `:value`, `:key`, `:both`, `:one` and `:either`.
- `single-thread` A generalized boolean.
**free-function**  
A designator for a function of two arguments.

**Description**  
The standard definition of `make-hash-table` is extended such that `test` can be any suitable user-defined function, except that it must not call `process-wait` or similar mp package functions which suspend the current process. If `test` is not one of the standard test functions (eg, `eql`, `equal` and `equalp`), and if `hash-function` is not supplied, then the hash value is the same as would be used if `test` were `equalp`.

`hash-function` may be supplied only if `test` is not one of the standard test functions. It takes a hash key as its argument and returns a hash value to use for hashing.

If `weak-kind` is non-nil, it makes `hash-table` weak. Its semantics are the same as the second argument of `set-hash-table-weak`, that is:

```
(make-hash-table :weak-kind weak-kind <other-args>)
```

is equivalent to

```
(let ((ht (make-hash-table <other-args>)))
  (set-hash-table-weak ht weak-kind)
  ht)
```

The default value of `weak-kind` is `nil`.

`single-thread`, if true, tells `make-hash-table` that the table is going to be used only in single thread contexts, and therefore does not need to be thread-safe. Single thread context means that only one thread can access the table at any point in time. That may be because the table is used only in one thread, but it can also be the case if the table is only ever accessed in the scope of a `lock`. Making a table with `single-thread` makes access to this table faster, but not thread-safe. It does not have other effects. The default value of `single-thread` is `nil`.

`free-function` adds a "free action" for a weak hash table. This has an effect only if `make-hash-table` is called with `weak-kind` non-nil. The `free-function` is called after an entry is auto-
matically removed by the garbage collector (GC). If *weak-kind* is `nil`, *free-function* is ignored.

*free-function*, if supplied, must take two arguments: *key* and *value*. When an entry is removed from a weak table *hash-table* because the relevant object is not pointed by any other object, the *key* and the *value* are remembered. Some time later (normally short, but not well-defined) the *free-function* is called with *key* and *value* as its arguments.

*free-function* needs to be fast, to avoid delays in unexpected places. Otherwise there are no restrictions on what *free-function* does. In particular, it can keep the *key* or *value* alive by storing them somewhere.

When objects are removed from the table by explicit calls (*remhash*, *clrhash*, *(setf gethash)*), *free-function* is not called.

**Notes**

Objects are removed from the table when the GC has identified them as free. For long-lived objects, which normally get promoted to higher generations, that may be quite a long time after the last pointer to them has gone.

*free-function* can also be specified in a call to *set-hash-table-weak*.

**See also**

- *hash-table-weak-kind*
- *modify-hash*
- *set-hash-table-weak*
- *with-hash-table-locked*

“Freeing of objects by the GC” on page 165

### make-instance

**Generic Function**

**Summary**

Creates and returns a new instance of a class.

**Package**

`common-lisp`
Signature  

```
make-instance class &rest initargs &key &allow-other-keys => instance
```

Arguments  

- `class`: A class, or a symbol that names a class.
- `initargs`: An initialization argument list.

Values  

- `instance`: A fresh instance of class `class`.

Description  

`make-instance` behaves as specified in ANSI Common Lisp. In particular it checks the initialization arguments as calculated by `compute-class-potential-initargs`.

This check can be suppressed by passing `:allow-other-keys t`. In addition, LispWorks provides global control over the initarg checking via `set-clos-initarg-checking` and per-class control via `class-extra-initargs`.

Notes  

In a delivered image, `make-instance` does not check the initialization arguments.

Compatibility notes  

In LispWorks 4.2 and previous versions, `make-instance` does not check the initargs. If your code contains invalid initargs, you could use one of the techniques mentioned above to resolve it.

See also  

- `class-extra-initargs`
- `compute-class-potential-initargs`
- `set-clos-initarg-checking`

### make-sequence  

**Function**

Summary  

Extends the standard `make-sequence` function allowing it to take any type specifier.

Package  

`common-lisp`
Signature  \texttt{make-sequence result-type size \&key initial-element \Rightarrow sequence}

Arguments
\begin{itemize}
  \item \textit{result-type} A type specifier.
  \item \textit{size} A non-negative integer.
  \item \textit{initial-element} An object.
\end{itemize}

Values \texttt{sequence} A sequence.

Description
The function \texttt{make-sequence} has been extended to take any Common Lisp type. The \texttt{sequence} will be of type \texttt{result-type} unless this is not possible, in which case a \texttt{type-error} is signaled.

See also \texttt{concatenate} \texttt{map} \texttt{merge}

\textbf{make-string}

Function

Summary Creates and returns a string.

Package \texttt{common-lisp}

Signature \texttt{make-string size \&key initial-element element-type \Rightarrow string}

Arguments \begin{itemize}
  \item \textit{size} A non-negative integer.
  \item \textit{initial-element} A character. The default is implementation-dependent.
  \item \textit{element-type} A type specifier. The default is defined below.
\end{itemize}

Values \texttt{string} A string.
The function `make-string` behaves as specified in the ANSI Common Lisp Standard with one exception: the default value of `element-type` is the value of `*default-character-element-type*` or the type of `initial-element` if that is a supertype of `*default-character-element-type*`

Therefore for strict compliance you must call `set-default-character-element-type` to set the default string element type to `character`.

See also `*default-character-element-type*`  
`set-default-character-element-type`  
`simple-string`

### make-string-output-stream

**Function**

**Summary**

Creates a character output stream.

**Package**

`common-lisp`

**Signature**

`make-string-output-stream &key element-type => stream`

**Arguments**

`element-type`  
A type specifier.

**Values**

`stream`  
A string output stream.

**Description**

The function `make-string-output-stream` behaves as specified in the ANSI Common Lisp Standard with one exception: the default value of `element-type` is the value of `*default-character-element-type*`.

Therefore for strict compliance you must call `set-default-character-element-type` to set the default string type to `character`. 
See also
- with-output-to-string
- *default-character-element-type*
- set-default-character-element-type

**map**

**Summary**
Redefines the standard `map` function allowing it to take any type specifier.

**Package**
common-lisp

**Signature**
`map result-type function &rest sequences => result`

**Arguments**
- `result-type`
  A sequence type specifier or `nil`.
- `function`
  A function designator.
- `sequence`
  A sequence.

**Values**
- `result`
  A sequence.

**Description**
The function `map` has been extended to take any Common Lisp type. The `result` will be of type `result-type` unless this is not possible, in which case a `type-error` is signaled.

See also
- concatenate
- make-sequence
- merge

**merge**

**Summary**
Redefines the standard `merge` function allowing it to take any type specifier.

**Package**
common-lisp
Signature

merge result-type sequence1 sequence2 predicate &key key =>
sequence

Arguments

result-type  A type specifier.
sequence1  A sequence.
sequence2  A sequence.
predicate  A function designator.
key  A function designator or nil.

Values

sequence  A sequence.

Description

The function merge has been extended to take any Common Lisp type. The sequence will be of type result-type unless this is not possible, in which case a type-error is signaled.

See also

concatenate
make-sequence
map

open

Function

Summary

Creates, opens, and returns a file stream that is connected to a specified file.

Package

common-lisp

Signature

open filespec &key direction element-type external-format if-exists if-
does-not-exist => stream

Arguments

filespec  A file designator.
direction  If direction is :probe, external-format is ignored. The element type and external format of the returned stream are undefined.
| **element-type** | By default, the value of *default-character-element-type* (the ANSI standard default is `character`). |
| **external-format** | An external file format designator. By default, this is `:default`. |
| **if-exists** | What to do if the file stream already exists. The possible values for this are as in the ANSI standard. |
| **if-does-not-exist** | What to do if the file stream does not already exist. The possible values for this are as in the ANSI standard. |

**Values**

- `stream`: A file stream, or `nil`.

**Description**

If `external-format` has a name which is not `:default` and the parameters include `:eol-style`, it is used as is.

Otherwise, the system decides which external format to use via `guess-external-format`. By default, this finds a match based on the filename; or (if that fails), looks in the EMACS-style (-*-*) attribute line for an option called `encoding` or `external-format` or `coding`; or (if that fails), chooses from among likely encodings by analyzing the bytes near the start of the file. By default, it then also analyzes the start of the file for byte patterns indicating the end-of-line style, and uses a default end-of-line style if no such pattern is found. This behavior is configurable.

After the external-format has been determined, it is verified using `valid-external-format-p`; and an error is signaled if this check fails.

If `open` gets `:default` as its `element-type` arg, it chooses the type on the basis of the external format. If `open` gets an `element-type` other than `:default` and the direction is `:input` or `:io`, the argument must be a supertype of the type of characters produced by the external format; if the direction is `:out-
put or :io, it must be a subtype of the type of characters accepted by the external format; if it does not satisfy these requirements, an error is signaled.

Standard stream input and output functions for character and binary data generally work in the obvious way on a file-stream with element-type base-char, (unsigned-byte 8) or (signed-byte 8). For example, read-sequence can be called with a string buffer and a binary file-stream: the character data is constructed from the input as if by code-char. Similarly write-sequence can be called with a string buffer and a binary file-stream: the output is converted from the character data as if by char-code. Also, 8-bit binary data can be read from and written to a base-char file-stream.

All standard stream I/O functions except for write-byte and read-byte have this flexibility.

See also
*default-character-element-type*
guess-external-format
set-file-dates
valid-external-format-p

open-stream-p

Summary A generic function that determines if a stream has been closed.

Package common-lisp

Signature open-stream-p stream => result

Arguments stream A stream.

Values result A generalized boolean.
The COMMON-LISP Package

Description
The function open-stream-p is generic. The default method provided by the class fundamental-stream returns t if close has not been called on the stream.

See also
close
fundamental-stream

output-stream-p
Generic Function

Summary
A generic function that determines if an object is an output stream.

Package
common-lisp

Signature
output-stream-p stream => result

Arguments
stream A stream.

Values
result A generalized boolean.

Description
The predicate output-stream-p is implemented as a generic function. The default method returns t if stream is an output stream. If the user wants to implement a stream with no inherent directionality (and thus does not include fundamental-input-stream or fundamental-output-stream) but for which the directionality depends on the instance, then a method should be provided for output-stream-p.

Examples
There is an example in “Stream directionality” on page 400.

See also
fundamental-output-stream
input-stream-p
### proclaim

**Function**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Established a specified declaration in the global environment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>common-lisp</td>
</tr>
<tr>
<td>Signature</td>
<td>proclaim declaration-list =&gt; nil</td>
</tr>
<tr>
<td>Arguments</td>
<td>declaration-list A list of declaration forms to be put into immediate and pervasive effect.</td>
</tr>
<tr>
<td>Values</td>
<td>Returns nil.</td>
</tr>
</tbody>
</table>
| Description      | Unlike declare, proclaim is a function that parses the declarations in the list (usually a quoted list), and puts their semantics and advice into global effect. This can be useful when compiling a file for speedy execution, since a proclamation such as: (proclaim '(optimize (speed 3) (space 0) (debug 0))) means the rest of the file is compiled with these optimization levels in effect. Other ways of doing this are:  
  - use the :optimize option in defsystem to establish default optimization qualities for every member of the system, when compiled via compile-system.  
  - add appropriate declare declarations in every function in the file.  
As proclaim involves parsing a list of lists of symbols and is intended to be used a few times per file at most, its implementation is not optimized for speed - it makes little sense to use it other than at top level.  
**Note:** For a top-level call to proclaim or declaim, optimize declarations are omitted from the compiled binary file. This deviates from the ANSI Common Lisp Standard but is useful because you are unlikely to want to change settings outside... |
of that file. To make the global settings, you can call a function which calls `proclaim` (so it is not a top-level call).

See “Compiler control” on page 104 for a more extended description of the compiler optimize qualities.

**Examples**

```lisp
(proclaim '(special *fred*))
(proclaim '(type single-float x y z))
(proclaim '(optimize (safety 0) (speed 3)))
```

**Notes**

As `proclaim` involves parsing a list of lists of symbols and is intended to be used a few times per file, its implementation is not optimized for speed — it makes little sense to use it other than at top level.

Remember to quote the argument list if it is a constant list. 

```lisp
(proclaim (special x))
```

attempts to call function `special`.

Exercise caution if you declare or proclaim variables to be special without regard to the naming convention that surrounds their names with asterisks.

**See also**

`compile`

`compile-file`

`declare`

`declare`

**restart-case**

*Macro*

**Summary**

Evaluates a restartable form in a special dynamic environment.

**Package**

`common-lisp`

**Signature**

`restart-case restartable-form {clause} => result*`
clause ::= (case-name lambda-list [:interactive interactive-expression | :report report-expression | :test test-expression])

Description

The macro **restart-case** behaves as specified in the ANSI Common Lisp standard.

In addition to that specification, **report-expression** may be a form whose **car** is **list**. Such a form is evaluated when the restart is set up and is expected to return a list of a format string and format arguments. When the restart is asked to report, this is done by calling **format** on the stream, the format string and the format arguments. This is more efficient than specifying an equivalent function, because no function object is created.

---

**room**

Function

Summary

Print information about the state of internal memory and its management.

Package

**common-lisp**

Signature

**room** &optional \(x\)

Arguments

\(x\) One of **nil**, **t**, or the keyword **:default**. Additionally in 64-bit LispWorks only, \(x\) can be the keyword **:full**.

Values

**room** returns no values.

Description

This function provides statistics on the current state of the memory, including the amount of space currently allocated, and the amount available for allocation.

As outlined in the Common Lisp Hyperspec, the **room** function takes an optional argument which controls the level of detail it produces.
The output of `room` differs between 32-bit and 64-bit LispWorks, and is described separately below.

**Output of `room` in 32-bit LispWorks**

Given an argument of `nil`, a summary of the total allocation in the entire heap (in kilobytes) is produced. The “allocated” figure only represents the amount of space allocated in heap segments that are writable, as opposed to read-only segments that hold some of the system code such as the garbage collector (GC) itself. The free space figure covers all the free space in all segments. To obtain these values programmatically, call `room-values`.

When called without an argument, `room` additionally prints information on the distribution of space between the generations of the heap.

When called with argument `t`, a breakdown of allocation in the individual segments of each generation is produced. Each segment is identified by its start address in memory. For each segment there is a free space threshold (the “minimum free space”)—when the available space in the segment falls below this value, the GC takes action to attempt to free more space in this segment.

Two statistics about promotion are also reported on a per-segment basis: the number of sweeps that an object must survive in this generation before becoming eligible for promotion, and the total volume of objects that have survived for that long and are consequently awaiting promotion to the next generation. These statistics are not relevant for static segments, which are indicated as “static”.

`room` prints numbers in decimal format, except for the segment start addresses which it prints in hexadecimal format.
Output of room in 64-bit LispWorks

The last line of the output of `room` is always a line containing the total allocated amount (memory occupied by live objects) and the total size (memory that LispWorks has allocated from the OS) (the "total line"). Both numbers are given in decimal followed parenthetically by the same number in hexadecimal. Above the total line is information for each generation.

`(room nil)` does not print any information about generations.

`(room)` prints the amount allocated for each generation in decimal.

`(room :full)` prints for each generation the amount allocated, both in decimal and in hexadecimal, and then the allocated amount of each allocation type in which there is any allocation in this generation.

`(room t)` prints for each generation the same as `(room :full)`, followed by information about the segments in this generation. For each segment it prints the allocation type and the start and end address for allocation in this segment.

Note that information for segments does not correspond to the allocated size, because not all the area in the segment is currently allocated.

See "Memory Management in 64-bit LispWorks" on page 146 for a description of allocation types and segments.

Output of room in the Mobile GC

The last line of the output of `room` is always a line containing the total allocated amount (memory occupied by live objects) and the total size (memory that LispWorks has allocated from the OS) (the "total line"). Both numbers are given in decimal followed parenthetically by the same number in hexadecimal. Above the total line is information for each generation.
(room) and (room :default) prints the allocated and free sizes according to these types:

Cons: cons object only.
Other: All other objects, except static objects and large objects (> 1 MB).
Static: Static objects.
Large: Large objects (> 1 MB). Note: this threshold may change in the future, but it is fixed in the current version.

The Cons and Other segments are divided according to their generation and there may also some permanent segments (as a result of a call to make-current-allocation-permanent, make-object-permanent or make-permanent-simple-vector).

In addition, LispWorks also holds some reserved segments that are used during GC, and room prints the size of these too.

(room t) also prints the segments for each type. For each segment, it prints the start and end addresses (in hex), the allocated area, and whether there is a free "hole" in the middle of it. For the Large and Static segments, it also prints the generation number of each segment. Permanent Static and Large segments have generation number 3.

See “Mobile GC technical details” on page 154 for more technical details.

Notes:

The segments information is useful for debugging problems with memory management, but for analysis of application allocation (room :full) gives enough information. Especially for very large images, there are many segments, so the output of (room t) is very large and not so useful (except for debugging).
Examples

CL-USER 22 > (room nil)
Total Size 39424K, Allocated 32591K, Free 6461K

CL-USER 23 > (room)
Generation 0:  Total Size 4394K, Allocated 952K, Free 3433K
Generation 1:  Total Size 1397K, Allocated 795K, Free 589K
Generation 2:  Total Size 4292K, Allocated 2172K, Free 2111K
Generation 3:  Total Size 29009K, Allocated 28885K, Free 114K
Total Size 39424K, Allocated 32805K, Free 6247K

CL-USER 24 > (room t)
Generation 0:  Total Size 4394K, Allocated 1004K, Free 3382K
  Segment 2008EC80: Total Size 507K, Allocated 353K, Free 149K
    minimum free space 64K,
    Waiting promotion = 23K, sweeps before promotion =10
  Segment 222B4498: Total Size 3886K, Allocated 650K, Free 3232K
    minimum free space 0K,
    Waiting promotion = 51K, sweeps before promotion =2
Generation 1:  Total Size 1397K, Allocated 589K
  Segment 2070DC18: Total Size 68K, Allocated 64K, Free 0K
    minimum free space 3K,
    Waiting promotion = 0K, sweeps before promotion =4
  Segment 21D84498: Total Size 1088K, Allocated 613K, Free 470K
    minimum free space 0K,
    Waiting promotion = 0K, sweeps before promotion =4
  Segment 200528D8: Total Size 240K, Allocated 118K, Free 118K
    minimum free space 0K, static
Generation 2:  Total Size 4292K, Allocated 2172K, Free 2111K
  Segment 21E94498: Total Size 4224K, Allocated
2107K, Free 2111K
  minimum free space OK,
  Awaiting promotion = 0K, sweeps
before promotion =4
Segment 20E7DC18: Total Size 68K, Allocated 64K, Free 0K
  minimum free space 117K,
  Awaiting promotion = 0K, sweeps
before promotion =4
Generation 3: Total Size 29009K, Allocated 28885K, Free 112K
  Segment 2071EC90: Total Size 7547K, Allocated 7543K, Free 0K
  minimum free space 0K,
  Awaiting promotion = 0K, sweeps
before promotion =10
  Segment 20E8EC90: Total Size 15318K, Allocated 15201K, Free 112K
  minimum free space 0K,
  Awaiting promotion = 0K, sweeps
before promotion =10
  Segment 2010DC18: Total Size 6144K, Allocated 6139K, Free 0K
  minimum free space 3K,
  Awaiting promotion = 0K, sweeps
before promotion =10
Total Size 39424K, Allocated 32857K, Free 6195K

See also
find-object-size
room-values
total-allocation
“Guidance for control of the memory management system” on page 130

**short-float**

*Type*

**Summary**
A subtype of float.

**Package**
common-lisp

**Signature**
short-float
Description

A short float is an immediate object.

`short-float` is disjoint from `double-float` in all LispWorks implementations in version 5.0 and later.

`short-float` is disjoint from `single-float` in all 32-bit LispWorks implementations, version 5.0 and later. In 64-bit LispWorks `short-float` is the same type as `single-float`.

Compatibility notes

In LispWorks 4.4 and previous on Windows and Linux platforms, `short-float` is the same type as `double-float`.

See also

double-float
long-float
parse-float
single-float

**short-site-name**  
*Function*

Summary

Identifies the physical location of the computer.

Package

`common-lisp`

Signature

`short-site-name => description`

`(setf short-site-name) description => description`

Arguments

`description` A string or `nil`.

Description

The function `short-site-name` returns a string briefly identifying the physical location of the computer. This should be set using `(setf short-site-name)` when you configure your LispWorks image.

See also

`long-site-name`
The **simple-string** type.

**Package**
common-lisp

**Signature**
`simple-string length element-type`

**Arguments**
- `length` The length of the string (or *, meaning any).
- `element-type` The type of string element. The default is the value of `*default-character-element-type*` rather than `*`.

**Description**
The union of all simple string types as specified in the standard, but extended with an extra parameter: `element-type`.

```
(simple-string length element-type) means all simple string types whose element type is a subtype of `element-type`. That is:

(simple-string * base-char) = (vector base-char *)

(simple-string * lw:bmp-char) = (or (vector base-char *)
                                       (vector lw:bmp-char *))

(simple-string * character) = (or (vector base-char *)
                                   (vector lw:bmp-char *)
                                   (vector character *))
```

**Example**
```
CL-USER 235 > (lw:set-default-character-element-type 'base-char)
BASE-CHAR
CL-USER 236 > (concatenate 'simple-string "f" "o" "o")
"foo"
CL-USER 237 > (type-of *)
SIMPLE-BASE-STRING
```

**See also**
- `*default-character-element-type*`  
- `set-default-character-element-type`  
- `string`
single-float

Type

Summary
A subtype of float.

Package
common-lisp

Signature
gle single-float

Description
A single-float is an immediate object in 64-bit LispWorks,
A single-float is a boxed object in 32-bit LispWorks.
single-float is disjoint from double-float in all LispWorks implementations, version 5.0 and later.
single-float is disjoint from short-float in all 32-bit LispWorks implementations in version 5.0 and later. In 64-bit LispWorks single-float is the same type as short-float.

Compatibility
notes
In LispWorks 4.4 and previous on Windows and Linux platforms, single-float is the same type as double-float. However, there are distinct specialized array types (array single-float), with single precision, and (array double-float), with double precision.

See also
double-float
long-float
parse-float
short-float

software-type

Function

Summary
Identifies the Operating System.

Package
common-lisp

Signature
software-type => description
Values

- **description**
  A string or `nil`.

Description

The function `software-type` returns a string representing a generic name of the Operating System, or `nil` if this cannot be determined.

On Windows, `software-type` returns "Windows NT". For more detail, use `software-version`.

See also

- `software-version`

---

### software-version

**Function**

**Summary**

Identifies the version of the Operating System.

**Package**

`common-lisp`

**Signature**

`software-version => description`

**Values**

- **description**
  A string or `nil`.

**Description**

The function `software-version` returns a string giving the version of the Operating System, or `nil` if this cannot be determined.

On Microsoft Windows systems, `description` begins with the specific Operating System. For supported systems this is "Windows Vista", "Windows Server 2008", "Windows 7", "Windows Server 2008 R2", "Windows 8", "Windows 10", "Windows Server 2012" or "Some Windows NT derivative". This is followed by the version numbers (Major.Minor), build number and optionally service pack.

**Compatibility notes**

On older unsupported operating systems, the `description` commences with "Windows 95", "Windows 98", "Windows Millennium", "Windows NT", "Windows 2000" or "Windows XP".
Prior to LispWorks 7.1, the description begins with "Windows 8" when running on Windows 10.

Example

```
(software-version)
=>
"Windows 8: 6.2 (build 9200)"

(software-version)
=>
"Windows 7: 6.1 (build 7600)"

(software-version)
=>
"Windows Vista: 6.0 (build 6000)"
```

See also software-type

*standard-input*
*standard-output*
*trace-output*
*error-output*
*query-io*
*debug-io*

Variables

Summary These are bound globally to synonyms to the *background-* streams:

Package common-lisp

Initial value Synonyms to the *background-* streams.

Description The variables *standard-input*, *standard-output*, *trace-output*, *error-output*, *query-io* and *debug-io* are bound globally to synonyms to the various default streams as follows:

* *standard-input* - synonym to *background-input*
*standard-output*, *trace-output* and *error-output* - synonym to *background-output*

*query-io* and *debug-io* - synonym to *background-query-io*

See *background-input* for details.

See also *background-input*

---

**step**  
*Macro*

**Summary**  
Steps through the evaluation of a form.

**Package**  
common-lisp

**Signature**  
step form => result

**Arguments**  
form  
A form to be stepped and evaluated.

**Values**  
result  
The values returned by form.

**Description**  
step evaluates a form and allows you to single-step through it. You can include a call to step inside a tricky definition to invoke the stepper every time the definition is used. step can also optionally step through macros.

The commands shown below are available. When certain stepper variables (as described below) are set, some of these commands are not relevant and are therefore not available. Use :help to get a list of the commands.

:s n  
Step this form and all of its subforms (optional positive integer argument).

:st  
Step this form without stepping its subforms.
:su       Step up out of this form without stepping its subforms.
:sr       Return a value to use for this form.
:sq       Quit from the current stepper level.
:redo     Redo one of the previous commands.
:get      Get an item from the history list and put it in a variable.
:help     List available commands.
:use      Replace one form with another form in previous command and redo it.
:his      List the commands history.

The optional integer argument $n$ for :s means do :s $n$ times.

Note: step is a Listener-based form stepper. LispWorks also offers a graphical source-code Stepper tool. See the LispWorks IDE User Guide for details of that.

Examples  The following examples illustrate some of these commands.

USER 12 > (step (+ 1 (* 2 3) 4))
(+ 1 (* 2 3) 4) -> :s
 1 -> :s
 1
(* 2 3) -> :su
 6
 4 -> :s
 4
11
11

USER 13 > (defun foo (x y) (+ x y))
FOO
USER 14 > step (foo (+ 1 1) 2)
(FOO (+ 1 1) 2) -> :st
  (+ 1 1) -> :s
       1 -> :s
       1 -> :s
       2
       2 -> :s
       2
       4

USER 15 > :redo (STEP (FOO # 2))
(FOO (+ 1 1) 2) -> :s
  (+ 1 1) -> :s
       1 -> :s
       1
       2
       2 -> :s
       2
       (+ X Y) -> :s
          X -> :s
          2
          Y -> :s
          2
       4

You can interact when an evaluated form returns, by setting the variable *no-step-out* to nil. The prompt changes as shown below:

USER 36 > step (cons 1 2)
(CONS 1 2) -> :s
  1 -> :s
    1 = 1 <- :sr 3
  2 -> :s
    2 = 2 <- :sr 4
(CONS 1 2) = (3 . 4) <- :s
(3 . 4)

To allow expansion of macros, set the variable *step-macros* to t.
To step through the function calls in compiled code, set the variable `hcl:*step-compiled*` to `t`.

If required, the stepper can print out the step level: set the variable `*print-step-level*` to `t`, as shown in this session:

```lisp
USER 21 > (setq *print-step-level* t)
T
USER 22 > step (cons 1 2)
[1] (cons 1 2) -> :s
[2] 1 -> :s       1
[2] 2 -> :s
  2
 (1 . 2)
(1 . 2)
```

It is not advisable to try to step certain compiled functions, such as `car` and `format`. The variable `hcl:*step-filter*` contains a list of functions which should not be stepped. If you get deep stack overflows inside the stepper, you may need to add a function name to `hcl:*step-filter*`.

By default, the stepper uses the same printing environment as the rest of LispWorks (the same settings of the `*print-*` variables). To control the stepper printing environment independently, set the variable `hcl:*step-print-env*` to `t`.

The values of the variables `hcl:*step-print-*` are then used instead of the variables `*print-*`.

---

**stream-element-type**

Generic Function

Summary: Implements `stream-element-type` as a generic function.

Package: `common-lisp`

Signature: `stream-element-type stream => type`

Arguments: `stream` A stream.
### string

<table>
<thead>
<tr>
<th><strong>Type</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>The string type.</td>
</tr>
</tbody>
</table>

| **Summary** |
| The string type. |

| **Package** |
| common-lisp |

| **Signature** |
| string length element-type |

| **Arguments** |
|---|---|
| length | The length of the string (or *, meaning any). |
| element-type | The type of string element. The default is the value of *default-character-element-type* rather than *. |

---

### Values

<table>
<thead>
<tr>
<th><strong>Values</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
</tr>
</tbody>
</table>

| **Description** |
| The function `stream-element-type` is implemented as a generic function. Depending on the stream, a method should be defined for this generic function that returns the element type of the stream. |

Methods must be implemented for all subclasses of `buffered-stream`. Typically for character streams, the implementation can return the `array-element-type` of the buffer.

For the class `fundamental-character-stream` a default method is provided which returns `character`. A method should be defined for stream classes based on the `fundamental-binary-stream` class.

There is an example in “Recognizing the stream element type” on page 400.

| **See also** |
| buffered-stream |
| fundamental-binary-stream |
| fundamental-character-stream |
Description

The union of all string types as specified in the standard, but extended with an extra parameter: element-type.

\[(\text{string length element-type})\] means all string types whose element type is a subtype of element-type. That is:

\[
\begin{align*}
(\text{string } * \text{ base-char}) &= (\text{vector base-char } *) \\
(\text{string } * \text{ lw:bmp-char}) &= (\text{or (vector base-char } *)) \\
&\quad \quad \quad \quad \quad \quad (\text{vector lw:bmp-char } *)) \\
(\text{string } * \text{ character}) &= (\text{or (vector base-char } *)) \\
&\quad \quad \quad \quad \quad \quad (\text{vector lw:bmp-char } *)) \\
&\quad \quad \quad \quad \quad \quad (\text{vector character } *))
\end{align*}
\]

Example

CL-USER 235 > (lw:set-default-character-element-type 'base-char)
BASE-CHAR
CL-USER 236 > (concatenate 'string "f" "o" "o")
"foo"
CL-USER 237 > (type-of *)
SIMPLE-BASE-STRING

See also

*default-character-element-type*
set-default-character-element-type
simple-string

**time**

*Macro*

Summary

Determines the execution time of a form in the current environment.

Package

common-lisp

Signature

\[\text{time form } \Rightarrow \text{ result}\]

Arguments

\[\text{form} \quad \text{A form to be evaluated.}\]

Values

\[\text{result} \quad \text{The result of the evaluation of the form.}\]
Description

The **time** can be used to determine execution times. The macro evaluates the form **form** and returns its value **result**. **time** also prints some timing and size data: **user time**, **system time**, **elapsed time**, and the total amount of heap space allocated in executing the form (in bytes).

The **user time** printed is the time used by LispWorks or any code that it calls in a dynamic library.

The **system time** printed is the time used in the operating system kernel when it is doing work on behalf of the LispWorks process.

The **elapsed time** printed is the time you could in principle measure with a stopwatch.

If LispWorks is 100% busy throughout the execution of the code, then **user time + system time = elapsed time**.

Each of the times is printed as:

- **secs.micros** if less than 60 seconds
- **hours:minutes:secs.micros** if 60 seconds or more.

The timing and size data covers all stack groups, not just the one that invokes **time**.

Notes

1. Note that **time** itself uses a small, constant amount of heap space.

2. **time** measures all threads, so to test accurately for consing in **code** you need to do:

   ```lisp
   (sys:with-other-threads-disabled (time code))
   ```

   This is particularly important when using the LispWorks IDE. Do not use **with-other-threads-disabled** in your application code.
Examples

```lisp
CL-USER 7 > (time (loop for i below 3000000
                        sum (sqrt i)))
Timing the evaluation of (LOOP FOR I BELOW 3000000 SUM (SQRT I))

User time = 0:01:04.187
System time = 0.062
Elapsed time = 0:01:07.297
Allocation = 4932022956 bytes
0 Page faults
Calls to %EVAL 72000048
3.4606518E9
```

See also
- extended-time
- with-other-threads-disabled
- with-unique-names
- “Guidance for control of the memory management system” on page 130

trace

**Macro**

**Summary**
Invoke the Common Lisp tracing facility on the named functions.

**Package**
common-lisp

**Signature**
```
trace {function-name|tracing-desc}* => trace-result
```
```
tracing-desc ::= (function-dspec {keyword form})*
```
```
  keyword ::= :after | :allocation | :before | :backtrace |
  :eval-after | :eval-before | :break |
  :break-on-exit | :entrycond | :exitcond |
  :inside | :process | :trace-output | :step |
  :when
```
```
qualifier ::= :after | :before | :around
```
```
function-name ::= symbol | (setf symbol)
```
The COMMON-LISP Package

Arguments

function-name  A symbol whose symbol-function is to be traced, or a setf function name. Functions, macros and generic functions may be specified this way.

function-dspec  A function-dspec as described in “Function dspecs” on page 84, which apart from symbols, can specify methods, setf functions and subfunctions.

tracing-desc  Specifies the functional definition which is to be traced and specifies any additional options that are required.

:after  is followed by a list of forms; these are evaluated upon returning from the function. The values of these forms are also printed out by the tracer. The forms are evaluated after printing out the results of the function call, and if they modify hcl:*traced-results* then the values received by the caller of the function are correspondingly altered (see also hcl:*traced-results*).

:allocation  — if non-nil, the memory allocation made during a function-call is printed upon exit from the function. This allocation is counted in bytes. If it is any other symbol (except nil), trace uses the symbol to accumulate the amount of allocation made between entering and exiting the function. Upon exit from the function, the symbol contains the number of bytes allocated during the function-call. For example,

(trace (print :entrycond nil
            :exitcond nil
            :allocation $$print-allocation))

results in $$print-allocation containing the sum of the allocation made inside print.

Note that if the function is called again, trace continues to use $$print-allocation as an accumulator of memory allo-
cation. It adds to the present value rather than re-initializing it each time the function is called.

`:backtrace` generates a backtrace on each call to the traced function. It is followed by a keyword that can be any of the following values:

`:quick` Like the `:bq` debugger command.
`t` Like the `:b` debugger command.
`:verbose` Like the `:b :verbose` debugger command.
`:bug-form` Like the `:bug-form` debugger command.

`:before` is followed by a list of forms; these are evaluated upon entering the function and their values are printed out by the tracer. The forms are evaluated after printing out the arguments to the function, and if they alter `*traced-arglist*` then the values received by the body of the function are changed accordingly (see also `*traced-arglist*`).

`:eval-after` and `:eval-before` are similar to `:after` and `:before`, without output.

`:break` is followed by a form. This is evaluated after printing the standard information caused by entering the function, and after executing any `:before` forms; if it returns `nil` then tracing continues normally, otherwise `break` is called. This provides a way of entering the debugger through the tracer.

`:break-on-exit` is followed by a form. This is evaluated after printing the standard information caused by returning from the function, and before executing any `:after` forms; if it returns `nil` then tracing continues normally, otherwise `break` is called. This provides a second way of entering the debugger through the tracer.

`:entrycond` controls the printing of the standard entry message (including the function’s arguments). If the form following it evaluates to give a non-nil value when the function is entered, then the entry message is printed (but otherwise it is not). If this option is not present then the standard entry mes-
sage is always printed upon calling the function. See also the
:when option.

:exitcond controls the printing of the standard exit message
(including the function's results). If the form following it
evaluates to give a non-nil value when the function is exited,
then the exit message is printed (but otherwise it is not). If
this option is not present then the standard exit message is
always printed upon returning from the function. See also
the :when option.

:inside restricts the tracing to within one of the functions
given as an argument. A single symbolic function name is
treated as a list of one element. For example, :inside for-
mat is equivalent to :inside (format).

:process may be used to restrict the tracing to a particular
process. If it is followed by a process then the function is only
traced when it is invoked from within that process. If it is fol-
lowed by t then it is traced from all processes — this is the
default. In any other cases the function is not traced at all.

:trace-output should be followed by a stream. All the out-
put from tracing the function is sent to this stream. By default
output from the tracer is sent to *trace-output*. Use of this
argument allows you to dispatch traced output from differ-
ent functions to different places.

:step, when non-nil, invokes the stepper (for evaluated
functions).

:when overrides all other keywords. It is followed by an
expression, and tracing only occurs when that expression
evaluates to non-nil. It is useful if you want to combine
:entrycond and :exitcond.

Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace-result</td>
<td>If trace is called with no arguments then it returns a list of the names of all the functions currently being traced. When called with</td>
</tr>
</tbody>
</table>
one or more arguments, it returns the symbols of the functions specified in those arguments.

Description  

trace is the macro used to invoke the tracing facility. This is a useful debugging tool that enables information about selected calls to be generated by the system. The standard way of invoking trace is to call it with the names of the functions, macros and methods that are to be monitored in this way. Calls to these produce a record of the function that was called, the arguments it received and the results it produced.

The arguments to trace each specify a function (or a macro or a method) to be traced. They may also contain further instructions to control how the tracing output is displayed, or to cause particular actions to occur when the functions is called or exited. If trace is called with a function that is already being traced, then the new tracing specification for that function replaces the old version.

Notes

For detailed information about the current tracing state, call tracing-state.

For information about problems with tracing and their resolution, see “Troubleshooting tracing” on page 60.
Example 1

USER 1 > (defvar *number-of-calls-to-max* 0)
*NUMBER-OF-CALLS-TO-MAX*

USER 2 > (trace (max :after
   ((incf *number-of-calls-to-max*))))
(MAX)

USER 3 > (dotimes (i 2) (max i 1))
0 MAX > (0 1)
0 MAX < (1)
1
0 MAX > (1 1)
0 MAX < (1)
2
NIL

USER 4 > *number-of-calls-to-max*
2

USER 5 > (trace (max
   :entrycond
   (> (length compiler:*traced-arglist*)
      2)
   :exitcond nil))
(MAX)

USER 6 > (max 2 3 (max 4 5))
0 MAX > (2 3 5)
5

Example 2
This example illustrates the use of :inside.
Example 3
To trace a method:

```
(defun foo (x) x)
(trace ((method foo (t))))
```

Example 4
To trace a setf function:

```
defvar *a* 0

(defun (setf foo) (x y) (set y x))
(setf foo)
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
>> X : 42
>> Y : *a*
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
```

```
defvar *a* 0
(defun (setf foo) (x y) (set y x))
(setf foo) (*a*)
```

See also  
*disable-trace*
*max-trace-indent*
*trace-indent-width*
*trace-level*
trace-new-instances-on-access
trace-on-access
*trace-output*
*trace-print-circle*
*trace-print-length*
*trace-print-level*
*trace-print-pretty*
*trace-verbose*
*traced-arglist*
*traced-results*
tracing-enabled-p
tracing-state
untrace

**truename**  
*Function*

<table>
<thead>
<tr>
<th>Summary</th>
<th>Returns the truename of a pathname.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>common-lisp</td>
</tr>
<tr>
<td>Signature</td>
<td>truename  filespec =&gt; truename</td>
</tr>
<tr>
<td>Arguments</td>
<td>filespec</td>
</tr>
<tr>
<td></td>
<td>A pathname designator.</td>
</tr>
<tr>
<td>Values</td>
<td>truename</td>
</tr>
<tr>
<td></td>
<td>A fully-specified physical pathname.</td>
</tr>
<tr>
<td>Description</td>
<td>The function truename behaves as specified in ANSI Common Lisp. The returned value is a fully-specified pathname. Truenames are always fully-specified in LispWorks (this prevents them from ever being corrupted by</td>
</tr>
</tbody>
</table>
*default-pathname-defaults*). Note that this means that the paths returned by directory are always fully specified.

See also directory

untrace                      Macro

Summary                     Turns off the Common Lisp tracing facility on the named functions.

Package                     common-lisp

Signature                   untrace {function-name | method-desc}* => untrace-list

Arguments                   function-name  A symbol whose symbol-function is no longer to be traced.
                             method-desc    A method description, as described in the entry for trace. See trace for more details.

Values                      When called with no arguments, it returns the symbols of all functions currently being traced. When called with one or more functions as arguments, untrace returns a list of the symbols of those functions. Thus, in all situations, untrace returns a list of the symbols of those functions being untraced.

Description                 untrace is used to cease the tracing of functions. If it is called with no arguments then the tracing of all currently traced functions is stopped. If it is called with one or more symbols then the tracing of those functions is stopped. A warning is given if untrace is called with a function that is not being traced.

Examples                    USER 12 > (progn (untrace) (trace + - / *))
                             *

721
To untrace a method:

\( \text{trace} \)  
\( \text{untrace-new-instances-on-access} \)  
\( \text{untrace-on-access} \)

**update-instance-for-different-class**

**Generic Function**

**Summary**
As specified for Common Lisp, and locks the redefined instance.

**Package**
common-lisp

**Description**
The generic function *update-instance-for-different-class* behaves as specified for ANSI Common Lisp.

During the operation of updating the instance, including the call to *update-instance-for-different-class*, the redefined instance is locked against access. Any other process that tries to access the instance will hang until the operation finishes. Therefore your methods must avoid doing anything that may wait for another process which may access the instance, as this would cause a deadlock.

**See also**
update-instance-for-redefined-class
**update-instance-for-redefined-class**  
*Generic Function*

**Summary**  
As specified for Common Lisp, and locks the redefined instance.

**Package**  
common-lisp

**Description**  
The generic function update-instance-for-redefined-class behaves as specified for ANSI Common Lisp.

During the operation of updating the instance, including the call to update-instance-for-redefined-class, the redefined instance is locked against access. Any other process that tries to access the instance will hang until the operation finishes. Therefore your methods must avoid doing anything that may wait for another process which may access the instance, as this would cause a deadlock.

**See also**  
update-instance-for-different-class

**with-output-to-string**  
*Macro*

**Summary**  
Creates a character output stream, performs a series of operations that may send results to this stream, and then closes the stream.

**Package**  
common-lisp

**Signature**  
with-output-to-string (var &optional string-form &key element-type) declaration form => result

**Description**  
The macro with-output-to-string behaves as specified in the ANSI Common Lisp Standard with one exception: the default value of element-type is the value of *default-character-element-type*. 
Therefore for strict compliance you must call `set-default-character-element-type` to set the default string type to `character`.

**See also**

- `compile-file`
- `declare`
- `proclaim`
- `*default-character-element-type*`
- `set-default-character-element-type`
This chapter describes symbols available in the COMPILER package.
The compiler is discussed in detail in Chapter 9, “The Compiler”.

**deftransform**  
*Macro*

**Summary**: Defines a function that computes the expansion of a form.

**Package**: compiler

**Signature**:  
```lisp
(defun deftransform name transform-name lambda-list &body body =>
list-of-transforms)
```

**Arguments**  
- `name` A symbol naming the function to which the transform is to be applied.
- `transform-name` The symbol naming the transformation — it should be unique for the function being transformed — and provides a handle with which to redefine an existing transform.
**lambda-list**  This must match against the form being expanded before expansion is allowed to take place, in the sense that it must be valid to call a function with such a lambda list using the arguments supplied in the candidate-form for expansion.

**body**  The body of the expander function, the result of which replaces the original form (unless it evaluates to `compiler:%pass%`, in which case no transformation is applied).

**Values**  

**list-of-transforms**  A list of the names of transforms defined for the function, including the one just added.

**Description**  

`deftransform`, like `defmacro`, defines a function that computes the expansion of a form. Transforms are only used by the compiler and not by the interpreter. `deftransform` allows you to add to the optimizations performed by the compiler.

**Examples**

```lisp
(compiler:deftransform  +  +of-2  (x y) '(system::|+2| ,x ,y))

(compiler:deftransform  +  +of-many (x &rest y) '(system::|+2| ,x (+ ,@y)))))

;; now an expression like (+ a b c 4 5 7 e f)
;; compiles to use the binary version
;; of + (inlined by default),
;; rather than the full (slow) version of +

(compiler:deftransform  list  list-of-1 (x) '(cons ,x '()))

(compiler:deftransform  list  list-of-2 (x y) '(cons ,x (cons ,y '())))

;; save having to call list -
;; cons is inlined by default
```
(compiler:deftransform constant my-trans (x)
  (cond ((constantp x) x)
        ((consp x) '(quote ,(eval x)))
        (t 'compiler::%pass%)) ; give up if not a cons

(compile (defun three-list () (constant (list 1 2 3))))

;; the function three-list returns the
;; same list (1 2 3)
;; every time it is called...

The list-of-2 example returns

(LIST-OF-2 LIST-OF-1 COMPILER::LIST-TRANSFORM)

as its result, since a similar transform already exists in the
compiler, by the name compiler::list*-transform.

Notes
deftransform differs from defmacro in various ways:

The evaluation of the body can return compiler:%pass%
indicating that the form is not to be expanded (in other
words, the transform method has elected to give up trying to
improve the code).

The compiler only calls the expander function if the argu-
ments match the lambda list — macros are unconditionally
expanded.

There can be several deftransforms for the same symbol,
each having a different name. (The compiler calls each one in
turn until one succeeds. This repeats until they all pass, so
that the replacement form may itself be transformed.)

If a transform takes keyword arguments the compiler pre-
serves the correct order of evaluation.

A carelessly written deftransform may lead the compiler to
transform valid Common Lisp into incorrect code — there is
no semantic checking of the transform.

See also compile
compile-file
This chapter describes symbols available in the DBG package, used to configure the debugging information produced by LispWorks.

The debugger is discussed in detail in Chapter 3, “The Debugger”.

**close-remote-debugging-connection**

*Function*

**Summary**

Close the remote debugging connection.

**Package**

dbg

**Signature**

close-remote-debugging-connection connection

**Arguments**

connection A remote-debugging-connection.

**Description**

The function close-remote-debugging-connection performs all the close cleanup operations that were associated with connection by remote-debugging-connection-add-close-cleanup, closes the underlying stream and clears any other resources that connection is using.
close-remote-debugging-connection may be called on either side (IDE or client) and causes the other side of the connection to call close-remote-debugging-connection on its own side.

Notes
close-remote-debugging-connection is called automatically by LispWorks when the other side closes the connection. It is also called automatically after you call configure-remote-debugging-spec with :setup-default nil when the Debugger or Remote Listener window is closed.

During debugging, you probably do not need to worry about closing connections, but if you use the remote debugging connections for other purposes then you will probably need to ensure they are closed, because each connection uses a process as well as some other resources.

See also
“Remote debugging” on page 25
remote-debugging-connection-add-close-cleanup

configure-remote-debugging-spec

Function

Summary
Client side: Configure how LispWorks opens a connection for remote debugging on the client side.

Package
dbg

Signature
configure-remote-debugging-spec host &key port log-stream failure-function timeout open-callback name setup-default enable => host

Arguments
host A string specifying the IDE side hostname, or nil.
port An integer.
log-stream An output stream or nil.
**failure-function**  
nil or a function of two arguments: host and port.

**timeout**  
A non-negative real or nil.

**open-callback**  
nil or a function that takes one argument (a newly opened connection).

**name**  
Any object.

**setup-default**  
:delayed, nil or t.

**enable**  
nil or t.

**Values**  

**host**  
A string or nil.

**Description**  
The function `configure-remote-debugging-spec` tells LispWorks how to open a connection for remote debugging on the client side if there is no existing connection to use (see `set-remote-debugging-connection` for details). `configure-remote-debugging-spec` changes the global settings, unless it is called inside the dynamic extent of `with-remote-debugging-spec`, in which case its effects last until the exit from this extent (except for `setup-default` and `enable`, see below).

If `port`, `log-stream`, `failure-function`, `timeout`, `open-callback` or `name` are supplied then they configure the corresponding settings, otherwise the settings are not changed.

`host` and `port` configure the hostname and TCP port to connect to (see the `hostspec` and `service` arguments to `open-tcp-stream`). `host` can also be nil, which specifies that LispWorks must not try to open a debugging connection. The initial configured value of `port` is the value of `*default-ide-remote-debugging-server-port*`, which is 21101 initially.

`timeout` configure the timeout in seconds, or nil for indefinite. If a connection to the IDE side cannot be made within the specified timeout then remote debugging is not used.
log-stream configures logging for communication problems, including failure to open the connection (unless the configured value of failure-function is non-nil) and errors when communicating across the connection. Failure to open the stream can happen if the host is not ready, the communication is blocked or the network/host are overloaded. Later failures should not happen as long as the underlying operating system is not broken. When the configured value of log-stream is non-nil, LispWorks writes error messages to it. Otherwise (the default) LispWorks does not log errors when communicating across the connection.

When the configured value of failure-function is non-nil, it will be called with the values of host and port in case of failure to open the connection, instead of writing to log-stream. The initial value of failure-function is nil.

The configured value of name is used as the name of the connection. It affects how the connection object is printed and also the name of the Lisp process that handles communication for the connection. It initially defaults to "Remote debugging".

When the configured value of open-callback is non-nil, it is called after the new connection has been opened, with the new connection as its argument. The initial configured value of open-callback is nil.

setup-default and enable determine whether the new connection becomes the default, and when it should be used. Note that they always have a global effect, even when configure-remote-debugging-spec is called in the dynamic extent of with-remote-debugging-spec.

When setup-default is :delayed (the default if the configured open-callback is nil) or t, the new connection will become the default connection, by a call to set-default-remote-debugging-connection. If enable is non-nil (the default), then the enabling switch is turned on as if by calling set-remote-debugging-connection with argument t. As a
result, assuming no other calls to `set-remote-debugging-connection` or `set-default-remote-debugging-connection` are made, the new connection will be used in the future whenever a connection is needed. See `set-remote-debugging-connection` for more details. If `setup-default` is `:delayed`, then the connection will be opened the first time it is needed. If `setup-default` is `t`, then `configure-remote-debugging-spec` opens the connection immediately. Thus with the default arguments, the connection will be opened the first time it is needed, and will then be used whenever a connection is needed.

When `setup-default` is `nil` (the default if the configured `open-callback` is non-nil), the connection is not made a default. If it is created when entering the debugger or for a Remote Listener, then it will be closed automatically when exiting the debugger or closing the Listener. See `remote-inspect` for how it deals with connections.

**Notes**

For the connection to open successfully, the machine which is addressed by `host` must be listening for TCP connections on `port`. Normally that should happen as result of calling `start-ide-remote-debugging-server`, but you can reasonably easily write your own version of it if required.

Using `configure-remote-debugging-spec` requires the ability to use `open-tcp-stream`, which at OS level means using the C library function `connect`.

Sometimes it is easier to make the connection in the other direction. For example, the Android SDK allows you to redirect sockets from a host to the android device, by using `adb forward`, so when the Android device is the client side, it is easier to connect the other way, in which case you should use `start-client-remote-debugging-server` on the client side (instead of using `configure-remote-debugging-spec`), and call `ide-connect-remote-debugging` on the IDE side.
Opening a connection once and then re-using it is probably more efficient in most cases and also has the advantage that remote object handles remain valid. However, if opening a connection is relatively rare, using one-off connections removes the (quite small) overhead of keeping a connection open.

When you do not re-use the connection, the configured values are used each time you open a connection.

If you wish to open a connection yourself, then note that you cannot implement the delayed automatic opening that configure-remote-debugging-spec implements when setup-default is :delayed (the default) or nil. You can, however, implement the equivalent of :setup-default t (that is opening the connection before it is needed) by making the underlying stream yourself using open-tcp-stream or some other mechanism, using create-client-remote-debugging-connection to create the client connection and then using set-default-remote-debugging-connection, set-remote-debugging-connection and with-remote-debugging-connection as appropriate.

See also

with-remote-debugging-spec
start-ide-remote-debugging-server
start-client-remote-debugging-server
start-remote-listener
remote-inspect
“Remote debugging” on page 25

create-client-remote-debugging-connection
create-ide-remote-debugging-connection

Summary
Create a client or ide remote debugging connection (advanced).

Package dbg
create-client-remote-debugging-connection name &key socket stream ssl log-stream => client-side-connection

create-ide-remote-debugging-connection name &key socket stream ssl log-stream => ide-side-connection

Arguments

name A string.
socket A socket or nil.
stream An base-char stream stream opened for :io or nil.
ssl A SSL specification.
log-stream An output stream or nil.

Values

client-side-connection A client-remote-debugging.
ide-side-connection An ide-remote-debugging.

Description

The function create-client-remote-debugging-connection creates a client side remote debugging connection. The function create-ide-remote-debugging-connection creates an IDE side remote debugging connection.

Normally you would use the higher level interface functions start-client-remote-debugging-server or configure-remote-debugging-spec to open a client side connection, and start-ide-remote-debugging-server or ide-connect-remote-debugging to open an IDE side connection. The higher level functions call these functions to create the connection.

name specifies a name for the connection, but otherwise does not affect its behavior. On the IDE side it is used in the title of Remote Listener and Remote Debugger tools. On both sides it is used in the name of the process that handles communication across the connection.

Either socket or stream (but not both) must be non-nil.
If `socket` is non-nil, it must be a socket handle, like the one that `start-up-server` passes to its `function` argument or the socket that `accept-tcp-connections-creating-async-io-states`, when called with `:create-state nil`, passes to its `connection-function`. A `socket-stream` is created with this socket and used as the stream in the connection. `socket` defaults to `nil`.

If `stream` is non-nil, it must be an `base-char` stream opened for `:io` and it is used directly in the connection. Typically it will be a `socket-stream`, but that is not a requirement. `stream` defaults to `nil`.

If `ssl` is non-nil, then `attach-ssl` is called on the stream, passing `ssl` as the `:ssl-ctx` argument. Note that this will work only if the stream is a `socket-stream`. `ssl` defaults to `nil`.

`log-stream`, if non-nil, must be an output stream. The connection writes messages to it in situations when communication fails. `log-stream` defaults to `nil`.

For the connection to work, the other side of the socket or stream must be the opposite kind of connection, that is for a client side connection the other side needs to be an IDE connection and vice versa.

The "ownership" of `socket` or `stream` is transferred to the connection by these functions. That means that no further I/O operations are allowed on `socket` or `stream` by other code, and they must not be closed. They will be closed when the connection is closed.

The value of `client-side-connection` that is returned by `create-client-remote-debugging-connection` can be used as the `connection` argument to `with-remote-debugging-connection`, `set-remote-debugging-connection` or `set-default-remote-debugging-connection`, or as the `connection` keyword to `start-remote-listener` or `remote-inspect`. It is not remembered by LispWorks anywhere.
The value of `ide-side-connection` that is returned by `create-ide-remote-debugging-connection` is remembered by LispWorks, and is returned by `ide-list-remote-debugging-connections` or `ide-find-remote-debugging-connection` when appropriate. As a result, it may be used by any of the IDE side functions. It can also be passed explicitly to any of these functions. It is forgotten when it is closed.

Both `client-side-connection` and `ide-side-connection` can be manipulated by these functions:

- `close-remote-debugging-connection` closes it.
- `remote-debugging-connection-add-close-cleanup` adds a cleanup callback that is called when the connection is called.
- `remote-debugging-connection-peer-address` finds the peer address of the connection.
- `remote-debugging-connection-name` returns its name.
- `ensure-remote-debugging-connection` checks if it is still open.

See also

- `start-client-remote-debugging-server`
- `configure-remote-debugging-spec`
- `start-ide-remote-debugging-server`
- `ide-connect-remote-debugging`
- `close-remote-debugging-connection`
- `remote-debugging-connection-add-close-cleanup`
- `remote-debugging-connection-peer-address`
- `remote-debugging-connection-name`
- `ensure-remote-debugging-connection`

“Remote debugging” on page 25
### *debug-print-length*

**Variable**

**Summary**
Controls the number of object components printed in debugger output.

**Package**
dbg

**Initial value**
40

**Description**
This variable is used to control the number of components of an object which are printed during output from the debugger. If its value is a positive integer then the components up to that number are printed. If it is nil then all the parts of an object are shown.

**Examples**
USER 83 > (setq dbg::*debug-print-length* 3)

3

USER 84 > (aref
'(1 2 3 4 "Jenny" "cottage" "door")
2)

Error: (1 2 3 4 Jenny cottage door) must be an array
1 (abort) return to top loop level 0.

Type :c followed by a number to proceed

USER 85 : 1 > :v
Call to ARRAY-ACCESS:
Arg 0 (ARRAY): (1 2 3 ...)
Arg 1 (SUBSCRIPTS): (2)
Arg 2 (SET-P): NIL Arg 3 (VALUE): NIL

**Notes**
*debug-print-length* is an extension to Common Lisp.

### *debug-print-level*

**Variable**

**Summary**
Controls the depth to which nested objected are printed in debugger output.
Package 

dbg

Initial value 

4

Description 

dbg:*debug-print-level* controls the depth to which nested objects are printed during output from the debugger. If its value is a positive integer then components at or above that level are printed. By definition an object to be printed is considered to be at level 0, its components are at level 1, their subcomponents are at level 2, and so on. If dbg:*debug-print-level* is nil then objects are printed to arbitrary depth.

Example 

USER 89 > (setq dbg:*debug-print-level* 2)

2

USER 90 > (subseq 3 '(cat (dog) ((goldfish))
  (((hamster)))))

Error: Illegal START argument (CAT (DOG)
  ((GOLDFISH))
  (((HAMSTER))))

1 (abort) return to top loop level 0.

Type :c followed by a number to proceed

USER 91 : 1 > :v

Call to CHECK-START-AND-END :
Arg 0 (START): (CAT (DOG) (#) (#))
Arg 1 (END): NIL

Notes 

*debug-print-level* is an extension to Common Lisp.

*default-client-remote-debugging-server-port* 

Variable

Summary 

The default TCP port number for the client side remote debugging server.

Package 

dbg
Initial Value  
21102.

Description  
The value of the variable *default-client-remote-debugging-server-port* is the default port for start-client-remote-debugging-server and ide-connect-remote-debugging.

Note that if you change the port number on one side you must change it to the same number on the other side.

See also  
start-client-remote-debugging-server  
ide-connect-remote-debugging  
“Remote debugging” on page 25

*default-ide-remote-debugging-server-port*  

Variable

Summary  
The default TCP port number for the IDE side remote debugging server.

Package  
dbg

Initial Value  
21101.

Description

Description  
The value of the variable *default-ide-remote-debugging-server-port* is the default port for start-ide-remote-debugging-server and the initial default for configure-remote-debugging-spec and with-remote-debugging-spec.

Note that if you change the port number on one side you must change it to the same number on the other side.

See also  
start-ide-remote-debugging-server  
configure-remote-debugging-spec
ensure-remote-debugging-connection

**Function**

**Summary**
Ensures that an object is a working remote debugging connection.

**Package**
dbg

**Signature**

```lisp
ensure-remote-debugging-connection object => connection-or-nil
```

**Arguments**

- `object` An object.

**Values**

- `connection-or-nil` A `remote-debugging-connection` or nil.

**Description**
The function `ensure-remote-debugging-connection` checks that `object` is a remote debugging connection (either IDE or client side) and that it is opened, and if so returns it. Otherwise it returns nil.

**Notes**
The main purpose of `ensure-remote-debugging-connection` is to check that the connection is still open. Debugging connections may close unexpectedly when the other side closes or the Lisp image quits or the machine is shut down, or (less likely) if something is wrong with the underlying connection between the machines.

**See also**

- “Remote debugging” on page 25

**executable-log-file**

**Function**

**Summary**
Returns the default bug form log file.
The DBG Package

Package  
dbg

Signature  
executable-log-file => log-file

Values  
log-file  
A pathname.

Description  
The function executable-log-file returns the default bug form log file for the current executable, which is the default path for *hidden-packages.* The path is also user specific.

See also  
log-bug-form  
logs-directory

*hidden-packages*  
Variable

Summary  
A list of packages whose symbols should not be displayed in debugger output.

Package  
dbg

Initial value  
A list containing the dbg and conditions packages.

Description  
dbg:*hidden-packages* is used by the debugger. It should be bound to a list of package specifiers. If a package is included in the list then any symbols in it are not shown by the debugger. Thus during backtraces the call frames corresponding to functions in these packages are not displayed. This can be useful in restricting the debugger to particular areas.
Example

CL-USER 1 > unbound

Error: The variable UNBOUND is unbound.
  1 (continue) Try evaluating UNBOUND again.
  2 Return the value of :UNBOUND instead.
  3 Specify a value to use this time instead of evaluating UNBOUND.
  4 Specify a value to set UNBOUND to.
  5 (abort) Return to level 0.
  6 Return to top loop level 0.

Type :b for backtrace or :c <option number> to proceed.
Type :bug-form "<subject>" for a bug report template or
:? for other options.

CL-USER 2 : 1 > :b 3
Call to ERROR
Call to EVAL
Call to CAPI::CAPI-TOP-LEVEL-FUNCTION

CL-USER 3 : 1 > (push "COMMON-LISP" dbg:*hidden-packages*)
("COMMON-LISP" #<The COMPILER package, 3131/4096 internal, 41/64 external> #<The SYSTEM package, 6258/8192 internal, 1266/2048 external> "DBG" "CONDITIONS")

Notes

1. *hidden-packages* can be set to value by
   (set-debugger-options :hidden value)

2. *hidden-packages* is an extension to Common Lisp.

See also

set-debugger-options
**ide-attach-remote-output-stream**

**Function**

**Summary**  
IDE side: Create a stream on the client side (a "client stream") attached to an IDE side stream.

**Package**  
dbg

**Signature**  
`ide-attach-remote-output-stream stream &key connection => stream-remote-object`

**Arguments**  
- `stream`  
  An output stream.

- `connection`  
  An `ide-remote-debugging` or `nil`.

**Values**  
- `stream-remote-object`  
  A remote object handle.

**Description**  
The function `ide-attach-remote-output-stream` creates an output stream on the client side which is attached to `stream`, such that any output written to the output stream on the client side is sent to the IDE side and written to `stream`. The returned `stream-remote-object` is a remote object handle corresponding to a client side output stream.

`ide-attach-remote-output-stream` must be called on the IDE side.

`connection` can be used to specify which connection to use. If `connection` is `nil`, then `ide-find-remote-debugging-connection` is called to find a connection. See `ide-find-remote-debugging-connection` for more details about finding a connection.

`ide-attach-remote-output-stream` returns the same `stream-remote-object` if it is called again for the same `stream` and `connection`.

`stream-remote-object` is returned on the IDE side, but must be used on the client side, so you need to pass it to the client, normally by one of `ide-set-remote-symbol-value`, `ide-...
funcall-in-remote or ide-eval-form-in-remote. For example, you can call:

\[
\begin{align*}
\text{(dbg:ide-set-remote-symbol-value} & \quad \text{'*my-log-stream*} \\
\text{(dbg:ide-attach-remote-output-stream} & \quad \text{mp:*background-standard-output*)})
\end{align*}
\]

After this call, anything that is written to *my-log-stream* on the client side will appear in the background output in the IDE.

Notes

ide-funcall-in-remote and ide-eval-form-in-remote themselves use this mechanism to bind their output-stream around the evaluation/function call.

If there are any errors when writing to stream, they are reported to the log-stream of the IDE side connection (as specified by ide-connect-remote-debugging and start-ide-remote-debugging-server). Possible reasons for such errors are:

- Writing a non base-char to a base-char stream. stream-remote-object itself can handle any character, but if stream is a base-char stream then an error will be signaled if the client writes a non base-char into it.
- Writing to a file that fills the disk.
- Other I/O errors.

See also

ide-find-remote-debugging-connection
“Remote debugging” on page 25

ide-connect-remote-debugging

Function

Summary
IDE side: Connect to a client remote debugging server.

Package
dbg
Signature  
ide-connect-remote-debugging host &key port timeout open-a-listener name log-stream => connection

Arguments  
- **host**  A string.
- **port**  An integer.
- **timeout**  A non-negative real or nil.
- **open-a-listener**  A boolean.
- **name**  A string.
- **log-stream**  An output stream or nil.

Values  
connection  An ide-remote-debugging.

Description  
The function ide-connect-remote-debugging attempts to open a TCP stream to the client machine named by host on port number port. If this is successful within timeout seconds, then create-ide-remote-debugging-connection is called with the new stream, log-stream and a name constructed from name, host and a counter to create and return connection.

port defaults to the value of *default-client-remote-debugging-server-port*, which is 21102 initially.

timeout defaults to nil, which means waiting indefinitely (or until the operating system reports an error).

If open-a-listener is non-nil, ide-open-a-listener is called to open a Remote Listener. open-a-listener defaults to nil.

Notes  
The client machine (specified by host) must be accepting TCP connections on port number port, which would normally be done by calling start-client-remote-debugging-server. Normally you do not need to supply port because both start-client-remote-debugging-server and ide-connect-remote-debugging default it to the value of *default-client-remote-debugging-server-port*.
The underlying TCP stream functionality must be working between the machines, that is they must be able to connect by TCP.

When using the Remote Debugger, Remote Listener or Inspector, you do not need to access the connection directly because the tools make one for you. In addition, `create-ide-remote-debugging-connection` remembers the connection, so all the IDE side functions that look for connections will find it.

You could easily implement your own version of `ide-connect-remote-debugging` if needed using `open-tcp-stream` and `create-ide-remote-debugging-connection`.

The editor command Connect Remote Debugging calls `ide-connect-remote-debugging`.

See also

- `start-client-remote-debugging-server`
- `open-tcp-stream`
- “Remote debugging” on page 25

### Functions

**ide-eval-form-in-remote**

**ide-funcall-in-remote**

**ide-set-remote-symbol-value**

**Summary**
IDE side: Evaluate a Lisp form, call a function or set a variable, all on the client side.

**Package**
dbg

**Signature**

- `ide-eval-form-in-remote form &key encoded-result timeout connection output-stream force-simple => results`
- `ide-funcall-in-remote func-and-args &key encoded-result timeout connection output-stream => results`
- `ide-set-remote-symbol-value symbol value &key connection => value`
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>form</td>
<td>A Lisp form.</td>
</tr>
<tr>
<td>encoded-result</td>
<td>One of \texttt{nil}, \texttt{t}, \texttt{:symbols} or \texttt{:not-cons}.</td>
</tr>
<tr>
<td>timeout</td>
<td>A non-negative \texttt{real} or \texttt{nil}.</td>
</tr>
<tr>
<td>connection</td>
<td>An \texttt{ide-remote-debugging} or \texttt{nil}.</td>
</tr>
<tr>
<td>output-stream</td>
<td>An output stream.</td>
</tr>
<tr>
<td>force-simple</td>
<td>A boolean.</td>
</tr>
<tr>
<td>func-and-args</td>
<td>A list.</td>
</tr>
<tr>
<td>symbol</td>
<td>A symbol.</td>
</tr>
<tr>
<td>value</td>
<td>An object.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>results</td>
<td>The values returned by evaluating the form or calling the function, or the two values \texttt{nil} and \texttt{:timeout-waiting-for-remote}.</td>
</tr>
<tr>
<td>value</td>
<td>An object.</td>
</tr>
</tbody>
</table>

Description

The function \texttt{ide-eval-form-in-remote} evaluates \texttt{form} on the client side and returns the result(s). Evaluation is done by \texttt{(eval form)}, unless either the evaluator has been eliminated (delivery with \texttt{:keep-eval nil}) or \texttt{force-simple} is non-nil, in which case a (very) simple evaluator is used. The simple evaluator recognizes symbols and conses, and returns all other objects. For a symbol, it returns the symbol-value (not recognizing symbol-macros). For a cons, if the car is one of \texttt{quote}, \texttt{if} or \texttt{progn} then it uses the Common Lisp semantics (using itself for recursion), otherwise it uses itself to evaluate all the elements of the cons's cdr and applies the car to those values.

The function \texttt{ide-funcall-in-remote} applies the car of \texttt{func-and-args} to the cdr of \texttt{func-and-args} and returns the result(s). Note that it does not do any evaluation of the elements of \texttt{func-and-args}. 
The function `ide-set-remote-symbol-value` calls `ide-funcall-in-remote` to set the client side `symbol` to `value` as in:

```
(dbg:ide-funcall-in-remote '(set symbol value)
  :connection connection)
```

`encoded-result` affects how the results are represented, where an "encoded" object is represented as a remote object handle and an "unencoded" object is represented as a normal object on the IDE side. Numbers, characters and strings are always unencoded, and all other objects except symbols and conses are always encoded. The value of `encoded-result` affects symbols and conses as follows:

- **nil** (default) The top-level conses are unencoded, which means that a result that is a list is returned as an unencoded list on the IDE side. Symbols that are in packages that exist on the IDE side are returned as unencoded symbols. Note that non top-level conses are encoded.
- **:symbols** The top-level conses are unencoded, but symbols and all other conses are encoded.
- **:not-cons** All conses, and symbols that are in packages that exist on the IDE side, are unencoded.
- **t** All conses and symbols are encoded.

`timeout` determines how long to wait for the result(s). If the client side does not return result(s) within `timeout` seconds, then `ide-eval-form-in-remote` and `ide-funcall-in-remote` return the two values `nil` and `:timeout-waiting-for-remote`.

`connection` can be used to specify which connection to use. If `connection` is `nil`, then `ide-find-remote-debugging-connection` is called to find a connection. See `ide-find-remote-debugging-connection` for more details about finding a connection.
output-stream must be an output stream, and defaults to the value of *standard-output*. During the evaluation or call on the client side, the variables *standard-output*, *error-output* and *trace-output* are bound to a stream that sends anything that is written to it back to the IDE side where it is written to output-stream. This is done using ide-attach-remote-output-stream to create a client side stream.

Notes

form, func-and-args, symbol and value are printed on the IDE side and read on the client side, and therefore must be objects that can be printed and read correctly. If the client is shaken (for example, delivered at level 2 or more), some symbols may not exist on the client side and will cause errors. If you want to ensure that specific symbols will work correctly with these functions, then use deliver-keep-symbols to keep them in the client.

Any remote object handle inside any of form, func-and-args, symbol or value is replaced by the object itself on the client side.

See also

ide-find-remote-debugging-connection
“Remote debugging” on page 25

ide-find-remote-debugging-connection
ide-set-default-remote-debugging-connection
ide-list-remote-debugging-connections

Functions

Summary
IDE side: Find or set as default an IDE side remote debugging connection.

Package
dbg

Signature
ide-find-remote-debugging-connection => connection-or-nil, default-p
ide-set-default-remote-debugging-connection connection => connection

ide-list-remote-debugging-connections &optional match-string => connections

Arguments
connection An ide-remote-debugging.
match-string A string.

Values
connection-or-nil An ide-remote-debugging or nil.
default-p A boolean.
connection An ide-remote-debugging.
connections A list of ide-remote-debugging.

Description
The function ide-find-remote-debugging-connection tries to find a useful IDE remote debugging connection. If ide-set-default-remote-debugging-connection was called, and the connection argument in the last call is still open, then ide-find-remote-debugging-connection returns this connection as connection-or-nil and default-p is t. Otherwise it returns the connection that was opened last and is still open as connection-or-nil and default-p is nil. If there are no opened connections then it returns connection-or-nil and default-p both as nil.

The function ide-set-default-remote-debugging-connection sets the default connection that ide-find-remote-debugging-connection will return to connection. connection must be a valid IDE side remote debugging connection, that is an instance of ide-remote-debugging that is still open. You can obtain one by calling create-ide-remote-debugging-connection, ide-find-remote-debugging-connection, ide-list-remote-debugging-connections or remote-object-connection.

The function ide-list-remote-debugging-connections returns a list of opened connections. If match-string is nil, the list contains all the connections. If match-string is non-nil, the
list contains only the connections whose name contains
*match-string* as a substring (plain match, case-insensitive).

**Notes**

`ide-find-remote-debugging-connection` is used by all the IDE side remote debugging interface functions like `ide-open-a-listener` and `ide-eval-form-in-remote` when their `connection` argument is `nil` (the default).

The various Editor commands (starting with "Remote"), except those ending with "In Listener", use `ide-eval-form-in-remote`, and therefore also use `ide-find-remote-debugging-connection`. The Editor command Set Default Remote Debugging Connection uses `ide-list-remote-debugging-connections` and `ide-set-default-remote-debugging-connection`.

The Remote Debugger and Remote Listener tools, and all remote object handles, are each associated with a specific connection, and therefore do not use `ide-find-remote-debugging-connection`.

`ide-find-remote-debugging-connection` and `ide-list-remote-debugging-connections` can find the connections because `create-ide-remote-debugging-connection` remembers each connection it creates. The higher level interface functions `start-ide-remote-debugging-server` and `ide-connect-remote-debugging` use `create-ide-remote-debugging-connection`.

**See also**

`ide-open-a-listener`
`ide-eval-form-in-remote`
`ide-funcall-in-remote`
`ide-set-remote-symbol-value`
`ide-attach-remote-output-stream`
`create-ide-remote-debugging-connection`
`ide-connect-remote-debugging`
`start-ide-remote-debugging-server`

"Remote debugging" on page 25
ide-open-a-listener

Function

Summary
IDE side: Open a Remote Listener tool.

Package
dbg

Signature
ide-open-a-listener &key connection timeout => process-remote-object

Arguments
connection An ide-remote-debugging or nil.
timeout A non-negative real or nil.

Values
process-remote-object A remote object handle or nil.

Description
The function ide-open-a-listener opens a Remote Listener, by calling start-remote-listener on the client side of the connection.

If connection is non-nil, it is used, otherwise ide-open-a-listener calls ide-find-remote-debugging-connection to find a connection. See ide-find-remote-debugging-connection for further details.

timeout specifies the length of time (in seconds) to wait before returning. timeout defaults to 10 and nil means waiting indefinitely.

When successful, ide-open-a-listener returns process-remote-object, which is a remote object (see remote-object-p) corresponding to the client process object that runs the read-eval-print loop. When a timeout occurs, process-remote-object is nil.

Notes
When process-remote-object is nil it does not necessarily means a failure and the Remote Listener may open later anyway. In particular, you can use :timeout 0 to avoid any waiting.
log-bug-form

Function

Summary
Writes a log of an error. This is useful in an application’s error handlers.

Package
dbg

Signature
log-bug-form description &key log-file message-stream => path

Arguments
description A string.
log-file A pathname designator.
message-stream An output stream, t or nil.

Values
path A pathname.

Description
The function log-bug-form is a simple interface for writing a log of an error. Your application’s error handlers can call it.

log-bug-form opens the file log-file for output. It writes the current date followed by a bug form. The bug form contains description, and debugging information generated by the system. When it finishes it writes to the stream message-stream a single line reporting that a bug form was written.

If log-file is supplied it must be a valid path, and it is used to open the file. The default value of log-file is the value returned by executable-log-file.

log-bug-form calls ensure-directories-exist before opening the log file, therefore so the directory where the log-file is written does not need to exist before log-bug-form is called.

If message-stream is t the message is written to standard output. If message-stream is a stream the message is written to it,
and if `message-stream` is `nil` then no message is written. `message-stream` defaults to the value of `*error-output*`.

If there is an error during the operation, `log-bug-form` silently fails and returns `nil`.

On success `log-bug-form` returns the path where the log file was written.

See also the section "Reporting bugs" in the *LispWorks Release Notes and Installation Guide*.

**Notes**

`log-bug-form` is invoked automatically if the debugger decides to use the console (the terminal) rather than use the LispWorks IDE debugging tools. This means that after such an error the user can always find a bug form in the default log file, which can be found by using `executable-log-file`.

`log-bug-form` always appends, so if it is called frequently the log file grows continuously. You may need to clear it periodically. It may be a good idea to move the file rather than delete it, so a record of errors remains.

When editing the log file it should be noted that each bug form is preceded by the time it was written, and that the bug forms are in chronological order. That means that the interesting bug form is most often the last one in the file.

**Compatibility notes**

In LispWorks 7.0 and earlier, `message-stream` defaulted to the value of `*debug-io*`, but this was not documented.

**See also**

`executable-log-file`

`logs-directory`

---

**logs-directory**

**Function**

**Summary**

Returns the directory in which LispWorks puts log files.

**Package**

dbg
The function `logs-directory` returns the directory in which LispWorks puts log files for the current user.

See also `executable-log-file` `log-bug-form` `output-backtrace` `Function`

**output-backtrace**

**Summary**
Prints a backtrace of the current stack. For use in exception handling routines.

**Package**
`dbg`

**Signature**
`output-backtrace keyword &key stream printer-bindings`

**Arguments**
- `keyword` Defines how verbose the output should be. It can be one of `:quick`, `:brief`, `:verbose` or `:bug-form`, in increasing order of verbosity.
- `stream` An output stream designator.
- `printer-bindings` A list of conses.

**Description**
The function `output-backtrace` prints a backtrace of the current stack.

The output goes to the stream designated by `stream`.

`printer-bindings`, if supplied, must be a list of conses, where the car of each cons is a symbol. `printer-bindings` is ignored if `keyword` is `:quick`. Otherwise, around the actual printing it binds each symbol to the value in the cdr of the cons. This is
intended to override the bindings that are used in the functions that `output-backtrace` uses.

`output-backtrace` should be used by applications in their exception handling routines to log a backtrace whenever an unexpected situation arises. In general, any application that is not intended to be used by Lisp programmers should have error handlers to deal with unexpected situations, and all these handlers should use `output-backtrace`.

Notes

The symbols that can be bound are not limited to "printer" symbols, so the name `printer-bindings` is slightly misleading.

See also

`log-bug-form`

---

### `*print-binding-frames*` Variable

**Summary**

Controls whether binding frames are printed in debugger output.

**Package**

`dbg`

**Initial value**

`nil`

**Description**

This variable is used by the debugger when it displays the stack frames. Binding frames are formed when special variables are bound, but are normally not shown by the debugger. However if the value of `dbg:*print-binding-frames*` is true then the binding frames are shown.

**Notes**

1. `*print-binding-frames*` can be set to `value` by
   
   `(set-debugger-options :bindings value)`

2. `*print-binding-frames*` is an extension to Common Lisp.
Example

CL-USER 16 > (defun print-to-length (object length)
  (let ((*print-length* length))
    (prinnt object)))
PRINT-TO-LENGTH

CL-USER 17 > (setf dbg:*print-binding-frames* t)
T

CL-USER 18 > (print-to-length '(x y z) 2)
Error: Undefined operator PRINNT in form (PRINNT OBJECT).
  1 (continue) Try invoking PRINNT again.
  2 Return some values from the form (PRINNT OBJECT).
  3 Try invoking something other than PRINNT with the same arguments.
  4 Set the symbol-function of PRINNT to another function.
  5 Set the macro-function of PRINNT to another function.
  6 (abort) Return to level 0.
  7 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :? for other options

CL-USER 19 : 1 > :n print-to-length
Interpreted call to PRINT-TO-LENGTH

CL-USER 20 : 1 > :b :verbose 5
Interpreted call to PRINT-TO-LENGTH:
  OBJECT         : (X Y Z)
  LENGTH         : 2
  *PRINT-LENGTH* : 2

Block environment contour:
Tag environment contour:
Function environment contour
Variable environment contour: ()

Tag environment contour:
Block environment contour:
Function environment contour
Variable environment contour: ()

Call to EVAL (offset 184)
  EXP : (PRINT-TO-LENGTH (QUOTE (X Y Z)) 2)
See also

*print-catch-frames*  

Summary Controls whether catch frames are printed in debugger output.

Package dbg

Initial value t

Description This variable is used by the debugger when it displays the stack frames. Catch frames are created when the special form `catch` is used. They are set up so that throws to the matching tag can be received. By default the debugger displays these frames, but if *print-catch-frames* is set to nil then the catch frames are no longer shown.
Notes

1. `*print-catch-frames*` can be set to `value` by
   `(set-debugger-options :catchers value)`

2. `*print-catch-frames*` is an extension to Common Lisp.

Examples

USER 17 > (setq dbg:*print-catch-frames* nil)
NIL
USER 18 > (defun catch-it ()
   (catch 'tag (throw-it) (print "Not caught")))
CATCH-IT
USER 19 > (defun throw-it ()
   (throw 'tag (break)))
THROW-IT
USER 20 > (catch-it)
break
   1 (continue) return from break.
   2 (abort) return to top loop level 0.

Type :c followed by a number to proceed

USER 21 : 1 > :b 5
Interpreted call to (DEFUN THROW-IT):
Call to *%APPLY-INTERPRETED-FUNCTION*:
Interpreted call to (DEFUN CATCH-IT):
Call to *%APPLY-INTERPRETED-FUNCTION*:
Call to %EVAL:

See also

set-debugger-options

*print-handler-frames*  Variable

Summary Controls whether handler frames are printed in debugger output.

Package `dbg`

Initial value `nil`
Description

This variable is used by the debugger when it displays the stack frames. Handler frames are created by error handlers (see “The stack in the debugger” on page 12), and are normally not shown by the debugger. However if *print-handler-frames* is set to t then the handler frames are displayed.

Notes

1. *print-handler-frames* can be set to value by
   (set-debugger-options :handler value)

2. *print-handler-frames* is an extension to Common Lisp.

Example

USER 162 > (setq lw:*print-handler-frames* t)
  T
USER 163 > (defun test (n)
  (handler-case (fn-to-use n)
  (type-error () (format t "~%Type error~%" 0)))
  TEST
USER 164 > (test #C(1 1))

Error: Undefined function: FN-TO-USE, with args
  (#C(1 1))

1 (continue) Call FN-TO-USE again
  2 (abort) return to top loop level 0.

Type :c followed by a number to proceed
USER 165: 1 > :b 10
Catch frame: (NIL)
Catch frame: #:|block-catcher-1854|
Call to *%UNDEFINED-FUNCTION-FUNCTION* :
Call to %EVAL :
Call to RETURN-FROM :
Call to %EVAL :
Call to EVAL-AS-PROGN :
Handler frame: ((TYPE-ERROR %LEXICAL-CLOSURE%
  (LAMBDA
    (CONDITIONS::TEMP)
    (GO #:|lambda-633|))
    ((#:|lambda-632|) (N . #))
    NIL ((#:|lambda-631|) (TEST))
    ((#:|lambda-633| # #))))
Catch frame: "<* Catch All Object *>
Call to LET :

See also set-debugger-options

*print-invisible-frames*  Variable

Summary  Controls whether invisible frames are printed in debugger output.

Package  dbg

Initial value  nil

Description  This variable is used by the debugger when it displays the stack frames.

Invisible frames are those for functions with hcl:invisible-frame declarations. These are normally not shown by the debugger. However if *print-invisible-frames* is true then these frames are displayed.

Notes  1. *print-invisible-frames* can be set to value by
   (set-debugger-options :invisible value)
2. *print-invisible-frames* is an extension to Common Lisp.

See also set-debugger-options

*print-open-frames*  

Variable

Summary Controls whether open frames are printed in debugger output.

Package dbg

Initial value nil

Description This variable is used by the debugger when it displays the stack frames. Open frames are made by the system and are normally not shown by the debugger. However if *print-open-frames* is set to t then the open frames are displayed. It is unlikely that you need to examine open frames: their use is connected with implementation details.

Examples

USER 52 > (setq dbg:*print-open-frames* t)
T
USER 53 > (car 2)
Error: Cannot take CAR of 2
1 (abort) return to top loop level 0.

Type :c followed by a number to proceed

USER 54 : 1 > :b 3
Open frame (5)
Open frame (5)
Call to CAR-FRAME :

Notes *print-open-frames* is an extension to Common Lisp.
**print-restart-frames**

**Variable**

**Summary**
Controls whether restart frames are printed in debugger output.

**Package**
dbg

**Initial value**
nil

**Description**
This variable is used by the debugger when it displays the stack frames. Restart frames are formed when restarts are established (see “The stack in the debugger” on page 12), but are normally not shown by the debugger. However if *print-restart-frames* is set to t then the restart frames are shown.

**Example**
USER 43 > (setq dbg:*print-restart-frames* t)
T
USER 44 > (truncate 12.5 0.0)
Error: Division-by-zero caused by TRUNCATE of (12.5 0.0)
  1 (continue) Return a value to use
  2 Supply new arguments to use
  3 (abort) return to top loop level 0.
Type :c followed by a number to proceed
USER 45 : 1 > :b 5
Restart frame: (ABORT)
Catch frame: (NIL)
Catch frame: #:|block-catcher-3223|
Call to DIVISION-BY-ZERO-ERROR :
Call to TRUNCATEANY :
USER 46 : 1 >

**Notes**
1. *print-restart-frames* can be set to value by
   (set-debugger-options :restarts value)
2. *print-restart-frames* is an extension to Common Lisp.
remote-debugging-connection
client-remote-debugging
ide-remote-debugging

System Classes

Summary
Classes of the connections in the remote debugging APIs.

Package
dbg

Superclasses
t

Description
The system class remote-debugging-connection is a super-class of all the connection classes in the remote debugging APIs. client-remote-debugging and ide-remote-debugging are subclasses of remote-debugging-connection.

All client side connections are instances of client-remote-debugging, and all IDE side connections are instances of ide-remote-debugging.

You should not try to instantiate these classes or inherit from them. In typical use, you do not need to access instances of these classes.

See “Remote debugging” on page 25 for how to create and use remote debugging connections.

See also
“Remote debugging” on page 25

remote-debugging-connection-add-close-cleanup
remote-debugging-connection-remove-close-cleanup

Functions

Summary
Add or remove a function that is called when a remote debugging connection is closed.
Package    dbg

Signature    remote-debugging-connection-add-close-cleanup  connection  
              function  =>  changed-p

               remote-debugging-connection-remove-close-cleanup  connection  
              function  =>  changed-p

Arguments    connection  A remote-debugging-connection.
              function  A function designator or a list whose car is a 
                           function designator.

Values       changed-p  Boolean.

Description  The function remote-debugging-connection-add-close- 
              cleanup records function as a cleanup in connection. When 
              connection is closed, for whatever reason, each recorded func-
              tion is invoked (by funcall for a function or symbol, or by 
              applying the car to the cdr for a list). function is added only 
              if it is not already in the list (tested by equal).

              The function remote-debugging-connection-remove-close- 
              cleanup removes function from the cleanups if it was 
              already added (tested by equal).

              changed-p is t if the cleanups were modified and nil other-
              wise.

              Both functions may be called on either side (IDE or client).

Notes        You should not assume anything about the order of calls to 
              the cleanup functions.

              Unhandled errors during the call to function are handled and 
              reported to the log-stream of the connection.

              remote-debugging-connection-remove-close-cleanup is needed when 
              you repeatedly create some objects that do not live for long 
              but you still want cleanups for them. In this 
              situation, the cleanup list would grow indefinitely unless you
call remote-debugging-connection-remove-close-cleanup when an object is discarded.

See also  “Remote debugging” on page 25
close-remote-debugging-connection

remote-debugging-connection-name  

*Function*

**Summary**
Return the name of a remote debugging connection.

**Package**
dbg

**Signature**
remote-debugging-connection-name connection => name

**Arguments**
connection  A remote-object-connection.

**Values**
name  An object.

**Description**
The function remote-debugging-connection-name returns the name was supplied (or defaulted) when connection was made.

remote-debugging-connection-peer-address may be called on either side (IDE or client).

See also  ide-connect-remote-debugging
start-ide-remote-debugging-server
create-ide-remote-debugging-connection
create-client-remote-debugging-connection
configure-remote-debugging-spec
“Remote debugging” on page 25
**remote-debugging-connection-peer-address**  
*Function*

**Summary**
Return the address of the other side of a remote debugging connection.

**Package**
dbg

**Signature**
`remote-debugging-connection-peer-address connection => remote-host, remote-port`

**Arguments**
- `connection` A remote-object-connection.

**Values**
- `remote-host` A string.
- `remote-port` An integer.

**Description**
The function `remote-debugging-connection-peer-address` returns the "peer address" of `connection`.

Normally debugging connections are implemented over socket streams, so `remote-host` and `remote-port` are the results of calling `socket-stream-peer-address` on the underlying stream.

`remote-debugging-connection-peer-address` is implemented by calling `remote-debugging-stream-peer-address` on the stream of the connection, which has a the method specialized on `socket-stream` that calls `socket-stream-peer-address`.

If `connection` does not use a socket stream (for example, if it was created by `create-ide-remote-debugging-connection` or `create-client-remote-debugging-connection` with a `stream` that is not a `socket-stream`), then `remote-host` and `remote-port` will both be `nil` unless you define a method on `remote-debugging-stream-peer-address` for the stream. That will not affect the behavior of the connection otherwise.
remote-debugging-connection-peer-address may be called on either side (IDE or client).

See also remote-debugging-stream-peer-address
“Remote debugging” on page 25

remote-debugging-stream-peer-address  

Generic Function

Summary  
Advanced: Return the peer address of a remote debugging stream.

Package  
dbg

Signature  
remote-debugging-stream-peer-address stream => remote-host, remote-port

Method signatures  
remote-debugging-stream-peer-address (stream t)
remote-debugging-stream-peer-address (stream socket-stream)

Arguments  
stream A stream.

Values  
remote-host A string.
remote-port An integer.

Description  
The generic function remote-debugging-stream-peer-address returns the "peer address" of stream as two values:

remote-host The hostname of the remote host.
remote-port The port number on the remote host.

remote-debugging-stream-peer-address is called by remote-debugging-connection-peer-address with the stream that its connection uses to communicate with the other side. It is intended to allow you to implement remote debugging with other types of stream by calling create-client-remote-debugging-connection and create-ide-remote-

769
 debugging-connection. In typical usage you do not need to define a method on remote-debugging-stream-peer-address.

The method specialized on \texttt{t} returns \texttt{nil} and \texttt{nil}.

The method specialized on \texttt{socket-stream} returns the peer hostname and port using \texttt{socket-stream-peer-address}.

remote-debugging-stream-peer-address may be called on either side (IDE or client).

See also remote-debugging-connection-peer-address
“Remote debugging” on page 25

---

remote-inspect

Function

**Summary**
Client side: inspect a client side object in an Inspector tool on the IDE side.

**Package**
dbg

**Signature**
remote-inspect object &key connection

**Arguments**

- \texttt{object} 
Any object.

- \texttt{connection} 
A client-remote-debugging.

**Description**
The function \texttt{remote-inspect} causes an Inspector tool on the IDE side to inspect a client side object. The actual object in the Inspector is a remote object handle to the client side \texttt{object}. The Inspector tool itself is an ordinary Inspector tool, and there is nothing that makes it a "remote" tool in any way.

\texttt{connection} specifies which connection to use. If it is supplied and is a valid open client connection, it is used.

If \texttt{connection} is \texttt{nil} (the default) or is not a valid connection, \texttt{remote-inspect} first checks if there is a default connection.
that is enabled (by default, if there is a default connection then it is enabled, see `set-remote-debugging-connection`) and uses that, the same as the Debugger and Listener do.

However, if there is no default connection enabled, `remote-inspect` behaves differently from the Debugger and Listener, because there is no obvious time-point to close temporary connections.

If there is no enabled default connection, `remote-inspect` does the following:

- If a previous call to `remote-inspect` already opened a connection, then `remote-inspect` re-uses it.
- Otherwise, if there a default connection then `remote-inspect` uses it anyway (even though it is not enabled).
- Otherwise, if there is a remote debugging spec (configured either by `configure-remote-debugging-spec` or `with-remote-debugging-spec`), then `remote-inspect` tries to open a connection using that spec. If this works, it uses the new connection, and unless it is configured as the default (setup-default non-nil in `configure-remote-debugging-spec`) it also records it for future calls of `remote-inspect`.

**Notes**

With the default setting, the connection opening function (`start-client-remote-debugging-server` or `configure-remote-debugging-spec`) both configures and enables the default connection, so `remote-inspect` will just use that connection (maybe opening it the first time).

An ordinary inspector can inspect a remote object because the generic function `get-inspector-values` has a method that specializes on remote object handles to invoke `get-inspector-values` on the client side and return the results. Thus `remote-inspect` can work only if `get-inspector-values` works on the client side. which is not guaranteed when delivering an application at higher values of the `level` argument to `deliver`. 
The only way to close a non-default connection that was opened by `remote-inspect` is to terminate the process that runs it on either the IDE or client side.

You can also inspect a remote object from the Remote Debugger or Remote Listener.

See also

- `set-remote-debugging-connection`
- `start-client-remote-debugging-server`
- `configure-remote-debugging-spec`
- `get-inspector-values`
- “Remote debugging” on page 25

### Functions

#### remote-object-p

**Summary**
IDE side: Test for a remote object handle and return its connection.

**Package**
dbg

**Signature**

```lisp
remote-object-p object => boolean
remote-object-connection remote-object => connection
```

**Arguments**

- `object` Any object.
- `remote-object` A remote object handle.

**Values**

- `boolean` A boolean.
- `connection` An `ide-remote-debugging`.

**Description**
The function `remote-object-p` is a predicate, returning true if `object` is a remote object handle and false otherwise.

The function `remote-object-connection` returns the `ide-remote-debugging` that `remote-object` is associated with.
See also “Remote debugging” on page 25

**set-debugger-options**

*Function*

**Summary**
Sets debugger printing control variables.

**Signature**

```
set-debugger-options &key all bindings catchers hidden handler restarts invisible
```

**Arguments**

- `all` A generalized boolean.
- `bindings` A generalized boolean.
- `catchers` A generalized boolean.
- `hidden` A generalized boolean.
- `handler` A generalized boolean.
- `restarts` A generalized boolean.
- `invisible` A generalized boolean.

**Description**

A call to `set-debugger-options` allows you to set the various debugger printing control variables without having the inconvenience of setting each variable individually with a call to `setq` and without having to remember the names for each of the variables.

`all` affects the state of the debugger command `:all`.

The other arguments set the debugger printing control variables as listed below:

- `bindings` *print-binding-frames*
- `catchers` *print-catch-frames*
- `hidden` *hidden-packages*
- `handler` *print-handler-frames*
- `restarts` *print-restart-frames*
Notes

The call frames are always displayed, so there is no option to control that.

See also

set-debugger-options

**set-default-remote-debugging-connection**

**Summary**

Client side (advanced): Sets the client side default connection to use when remote debugging is enabled.

**Package**

dbg

**Signature**

set-default-remote-debugging-connection connection => return-connection

**Arguments**

connection A client-remote-debugging.

**Values**

return-connection The value of connection.

**Description**

The function set-default-remote-debugging-connection sets connection as the default connection for the remote debugging interface on the client side (used when entering the debugger and by calls to start-remote-listener and remote-inspect).

The default connection is used when the enabling switch is t, which it typically is because that is the default in start-client-remote-debugging-server and configure-remote-debugging-spec. The switch can also be set by set-remote-debugging-connection and in a dynamic scope by with-remote-debugging-connection and with-remote-debugging-spec.

See set-remote-debugging-connection for a discussion about the enabling switch.
Notes

In typical usage, you will not need to use `set-default-remote-debugging-connection`.

See also

`set-remote-debugging-connection`

“Remote debugging” on page 25

---

**set-remote-debugging-connection**

*Function*

Summary

Client side (advanced): Set the remote debugging connection to use on the client side.

Package

dbg

Signature

`set-remote-debugging-connection connection => res`

Arguments

`connection` nil, t or a `client-remote-debugging`.

Values

`res` The value of `connection` if it is valid.

Description

The function `set-remote-debugging-connection` sets an enabling switch controlling which remote debugging connection will be used by the remote debugging interface (entering the debugger, `start-remote-listener` or `remote-inspect`) on the client side.

If `connection` is t, then the default connection will be used (which may be set by `set-default-remote-debugging-connection`, `start-client-remote-debugging-server`, or `configure-remote-debugging-spec`). If `connection` is an instance of `client-remote-debugging`, then `connection` itself will be used. If `connection` is nil, then no connection is specified (the API may open one when needed).

The setting is global, unless it is called within the dynamic extent of `with-remote-debugging-connection`, in which case its effects last until the exit from this extent.
When entering the debugger, the connection is used when `invoke-debugger` is called and no hook blocks it. Possible hooks include `*debugger-hook*` and `with-debugger-wrapper`. For the Remote Listener and Inspector, the functions `start-remote-listener` and `remote-inspect` use the connection.

**Notes**

In typical usage, you will not need to explicitly call `set-default-remote-debugging-connection`, because `start-client-remote-debugging-server` and `configure-remote-debugging-spec` (with the default parameters) both set the default connection.

In all cases, if the debugger is invoked and there is no connection to use, LispWorks may open a connection if it was configured to do so by `configure-remote-debugging-spec` or `with-remote-debugging-spec`.

You can obtain a client remote debugging connection by using the `:open-callback` keyword with `start-client-remote-debugging-server` or `configure-remote-debugging-spec`.

**See also**

- `with-remote-debugging-connection`
- `set-default-remote-debugging-connection`
- `start-client-remote-debugging-server`
- `configure-remote-debugging-spec`
- “Remote debugging” on page 25

---

**start-client-remote-debugging-server**

**Function**

**Summary**

Client side: Start a remote debugging TCP server on the client side.

**Package**

dbg
Signature

```
(start-client-remote-debugging-server &key port open-callback log-stream setup-default enable => process)
```

Arguments

- **port**
  - An integer.
- **open-callback**
  - nil or a function of one argument: the connection.
- **log-stream**
  - An output stream or nil.
- **setup-default**
  - nil or t.
- **enable**
  - nil or t.

Values

- **process**
  - A process.

Description

The function `start-client-remote-debugging-server` starts a client side server for remote debugging, which will create a client side remote debugging connection when a remote machine connects to it. With the default settings, this remote debugging connection will become the default connection and will be enabled for re-use whenever a connection is needed (entering the debugger, or calls to `start-remote-listener` or `remote-inspect`).

The main operation of `start-client-remote-debugging-server` is calling `start-up-server` with `port` and a function that calls `create-client-remote-debugging-connection` to create a client for every connection to the server. `port` defaults to the value of `*default-client-remote-debugging-server-port*`, which is 21102 initially.

The other keyword arguments affect what else `start-client-remote-debugging-server` does.

If `open-callback` is non-nil, it is called with the each new connection that is created. `open-callback` defaults to nil.

If `setup-default` is non-nil (the default when `open-callback` is nil), every new connection is made the default connection by a call to `set-default-remote-debugging-connection`, and if `enable` is non-nil (the default), it will be enabled for re-
use by \texttt{set-remote-debugging-connection} with argument \texttt{t}. If \texttt{setup-default} is \texttt{nil} (the default when \texttt{open-callback} is non-nil) then \texttt{enable} is ignored. \texttt{setup-default} defaults to \texttt{(not open-callback)}.

If \texttt{log-stream} is non-nil, LispWorks writes error messages to it relating to failures during communication on a connection. These failures should not happen normally, but may happen if something writes to the remote debugging connection not through the remote debugging interface. \texttt{log-stream} defaults to \texttt{nil}.

\texttt{start-client-remote-debugging-server} returns \texttt{process}, which is the result of \texttt{start-up-server}. You can use \texttt{server-terminate} to stop it.

\textbf{Notes}

To have a connection, the IDE side needs to connect to the client hostname on port number \texttt{port} using \texttt{ide-connect-remote-debugging}, or code that does similar things.

In normal operation, it is assumed that the IDE side will connect once, and from that point onward this connection will be used for all remote debugging. If the IDE subsequently opens another connection without first closing the first connection, then the first connection will "leak" on the client side. It is the responsibility of the IDE to close it in this case.

\texttt{open-callback} allows more complex usage, for example to store the connection somewhere and use it when required by \texttt{with-remote-debugging-connection} or \texttt{set-remote-debugging-connection}. Note that \texttt{setup-default} is \texttt{nil} by default when \texttt{open-callback} is non-nil.

\texttt{start-client-remote-debugging-server} is implemented using the other functions discussed in this section, so you can can reasonably easily write your own version of it if you need to.

\textbf{See also}

\texttt{start-up-server}
\texttt{create-client-remote-debugging-connection}
**set-default-remote-debugging-connection**
**set-remote-debugging-connection**
**ide-connect-remote-debugging**
“Remote debugging” on page 25

---

**start-ide-remote-debugging-server**  
*Function*

**Summary**  
IDE side: Start an IDE side remote debugging server, so clients can connect to it.

**Package**  
dbg

**Signature**  
`start-ide-remote-debugging-server &key port socket-filter name log-stream => server-process`

**Arguments**  
- `port`  
  An integer or string.
- `socket-filter`  
  A function of one argument or nil.
- `name`  
  A string.
- `log-stream`  
  An output stream or nil.

**Values**  
- `server-process`  
  A process handling incoming connections.

**Description**  
The function `start-ide-remote-debugging-server` starts a TCP server (by calling `start-up-server`) that creates IDE remote debugging connections.

- `port` must be an integer port number or string service name. It is supplied as the `service` argument to `start-up-server`. `port` defaults to the value of `*default-ide-remote-debugging-server-port*`, which is 21101 initially.

- `name` is used in the name of the created connections.

- If `log-stream` is non-nil, LispWorks writes error messages to it relating to failures during communication on a connection. These failures should not happen normally, but may happen if something writes to the remote debugging connection not
through the remote debugging interface. \textit{log-stream} defaults to \texttt{nil}.

If \textit{socket-filter} is non-nil, it is called with the connected socket before creating the connection, and if it returns \texttt{nil} then the socket is closed and no connection is created. \textit{socket-filter} defaults to \texttt{nil}.

The server creates IDE remote debugging connections by calling \texttt{create-ide-remote-debugging-connection} with the connected socket, a name (constructed from \texttt{name}, the peer address of the socket and a counter) and \textit{log-stream}.

\texttt{start-ide-remote-debugging-server} returns \texttt{server-process}, which is the first result of \texttt{start-up-server}. You can terminate the server by calling \texttt{server-terminate} with \texttt{server-process}.

\textbf{Notes}

The client would normally open a connection using \texttt{configure-remote-debugging-spec} with host specifying the machine on which \texttt{start-ide-remote-debugging-server} has been called with the same port as \texttt{port} (which defaults to the value of \texttt{*default-ide-remote-debugging-server-port*} for both functions).

The underlying TCP stream functionality must be working between the machines, that is they must be able to connect by TCP.

When using the Remote Debugger, Remote Listener or Inspector, you do not need to access the connection directly because the tools do it for you. In addition, \texttt{create-ide-remote-debugging-connection} remembers the connection, so all the IDE side functions that look for connections will find it.

\textit{socket-filter} can call \texttt{get-socket-peer-address} to check who is connecting. It can also be used just to log that a connection has been made, but must return \texttt{t} in this case.
You could easily implement your own version of start-ide-remote-debugging-server if needed using start-up-server and create-ide-remote-debugging-connection.

See also

- start-up-server
- create-ide-remote-debugging-connection
- “Remote debugging” on page 25

**start-remote-listener**

*Function*

**Summary**

Client side: Start a Remote Listener on the IDE side.

**Package**

dbg

**Signature**

```lisp
(start-remote-listener &key new-process-p message connection close-on-exit) => started-p
```

**Arguments**

- `new-process-p` A boolean, default `t`.
- `message` A string or `nil`.
- `connection` A `client-remote-debugging` or `nil`.
- `close-on-exit` A boolean.

**Values**

- `started-p` A boolean.

**Description**

The function `start-remote-listener` starts a Remote Listener tool on the IDE side, such that reading and evaluation is done on the client side where the `start-remote-listener` was called.

`start-remote-listener` first tells the IDE to start the Listener, and then runs a read-eval-print loop that communicates with the IDE's Listener over the connection.

If `new-process-p` is non-nil (the default), then a new Lisp process is created to start the Listener and run the read-eval-print loop. This process runs until the read-eval-print loop
exits. If new-process-p is \texttt{nil}, then the read-eval-print loop runs in the current process and \texttt{start-remote-listener} does not return until the read-eval-print loop exits.

\textit{message}, when is not \texttt{nil}, is printed into the Listener tool before the first prompt appears.

\textit{connection} (default \texttt{nil}) controls which connection to use. If \textit{connection} is non-nil and is connected then it is used. Otherwise \texttt{start-remote-listener} uses the same mechanism as the debugger to find the connection, which by default means re-using an existing connection if one exists, or opening a new one (under the control of the remote debugging spec). In typical usage, this be set up by either \texttt{configure-remote-debugging-spec} or \texttt{start-client-remote-debugging-server}. See “Simple usage” on page 26 and “Advanced usage - multiple connections” on page 33 for details.

\textit{close-on-exit} is used only when \textit{connection} is non-nil. When \textit{close-on-exit} is non-nil, the connection is closed when the read-eval-print loop exits. Otherwise (the default), the connection remains open for later re-use.

If the Listener tool on the IDE side is closed, then the read-eval-print loop exits. Normally this is the only way that the loop exits, but you could also exit it by throwing to a surrounding catch (when new-process-p is \texttt{nil}) or by terminating the process (by \texttt{current-process-kill}).

\texttt{start-remote-listener} returns \texttt{nil} if \textit{connection} is not a valid connection (either \texttt{nil} or already closed) and it cannot find the connection to use. Otherwise, if new-process-p is non-nil (the default) it returns \texttt{t} immediately, and if new-process-p is \texttt{nil} it returns \texttt{nil} only when the Listener is closed.

Notes

Using \textit{message} is an easy way for the client to write some text to the IDE even when you do not need a Listener.
See also set-remote-debugging-connection
configure-remote-debugging-spec
“Remote debugging” on page 25

*terminal-debugger-block-multiprocessing* Variable

Summary Controls blocking of multiprocessing in the terminal debugger.

Package dbg

Initial value t

Description When the debugger is entered on the terminal, multiprocessing is blocked if the value of *terminal-debugger-block-multiprocessing* is t. This is the default value.

If you set this variable to nil then other processes, including timers, will continue to run in parallel to the process that entered the terminal debugger (as they did before the debugger was entered). Beware that this will make it more difficult to debug multi-process activities.

The other allowed value is :maybe. This means that multiprocessing is blocked in the terminal debugger unless the debugger was entered from the CAPI environment.

The value of *terminal-debugger-block-multiprocessing* affects the behavior of a REPL started by start-tty-listener.

Example This listener session illustrates the effect of *terminal-debugger-block-multiprocessing*.

Firstly we see the default behavior whereby a call to print in another process is blocked by the debugger.
CL-USER 1 > dbg:*terminal-debugger-block-multiprocessing*
T

CL-USER 2 > unbound

Error: The variable UNBOUND is unbound.
1 (continue) Try evaluating UNBOUND again.
2 Specify a value to use this time instead of evaluating UNBOUND.
3 Specify a value to set UNBOUND to.
4 (abort) Return to level 0.
5 Return to top-level loop.
6 Return from multiprocessing.

Type :b for backtrace, :c <option number> to proceed, or :? for other options

CL-USER 3 : 1 > (setq *timer* (mp:make-timer 'print 10))
Warning: Setting unbound variable *TIMER*
#<Time Event : PRINT>

CL-USER 4 : 1 > (mp:schedule-timer-relative *timer* 1)
#<Time Event : PRINT>

CL-USER 5 : 1 > :a

On leaving the debugger the output 10 from the call to print appears. Then we set *terminal-debugger-block-multiprocessing* to nil and repeat the commands:
CL-USER 6>
10
(setf dbg:*terminal-debugger-block-multiprocessing* nil)
NIL

CL-USER 7> unbound

Error: The variable UNBOUND is unbound.
1 (continue) Try evaluating UNBOUND again.
2 Specify a value to use this time instead of evaluating UNBOUND.
3 Specify a value to set UNBOUND to.
4 (abort) Return to level 0.
5 Return to top-level loop.
6 Return from multiprocessing.

Type :b for backtrace, :c <option number> to proceed, or :? for other options

CL-USER 8:1> (setq *timer* (mp:make-timer 'print 10))
#<Time Event : PRINT>

CL-USER 9:1> (mp:schedule-timer-relative *timer* 1)
#<Time Event : PRINT>

CL-USER 10:1>
10

Notice above that the output 10 from the call to print appears after 1 second, in the debugger. Multiprocessing was not blocked.

See also start-tty-listener

with-debugger-wrapper

Macro

Summary
Executes code with a "debugger wrapper" which is called only if the debugger is invoked during the execution.

Package dbg
The macro `with-debugger-wrapper` executes forms in `body` with the function `wrapper` bound as a "debugger wrapper". This debugger wrapper takes effect only if the code in `body` tries to invoke the debugger (by a call to `invoke-debugger`), typically indirectly as a result of an error. Instead of entering the debugger, the debugger wrapper is called with two arguments: a function to call to enter the debugger, and the condition. The wrapper can do whatever is needed. If it wants to enter the debugger, it does it by calling its first argument with the second argument:

```
(funcall function condition)
```

Example

Suppose that you run many processes in parallel with the same code. If the code is broken then every process will get an error. This example shows how a debugger wrapper can be used to keep a `lock` around entry to the debugger, so that the processes enter the debugger one by one. It contains firstly the "application code", then the debugger wrapper, and lastly forms which execute the application with or without the debugger wrapper.
(in-package "CL-USER")

(defglobal-parameter *a* 0)

(defun foo (index cons)
  (sys:atomic-push (* index *a*) (cdr cons)))

;; This gets the process function so we can pass
;; the wrapper function instead.
(defun my-run-processes (do-error &optional
                      (process-function 'foo))
  (setq *a* (if do-error :do-error 7))
  (let ((cons (cons nil nil)))
    (dotimes (x 10)
      (mp:process-run-function
       (format nil "My test process -d* x")
       ()
       process-function
       x cons))
    (sleep 0.2)
    (print (cdr cons))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; debugger wrapper ;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun my-debugger-wrapper (func condition)
  (mp:with-lock (*my-debugger-lock*)
    (funcall func condition)))

(defun foo-wrapper (index cons)
  (dbg:with-debugger-wrapper 'my-debugger-wrapper
    (foo index cons)))

;; Running the application without the wrapper fills
;; your screen with notifiers
(my-run-processes t)

;; Running with the wrapper raises the notifiers one by
;; one. You can use the Process Browser kill them all.
(my-run-processes t 'foo-wrapper)
See also “Debugger troubleshooting” on page 23

### with-remote-debugging-connection

#### Summary
Client side (advanced): Dynamically bind the remote debugging connection to use on the client side.

#### Package
dbg

#### Signature
```lisp
(with-remote-debugging-connection (connection) &body body => body-values)
```

#### Arguments
- `connection`: nil, t or a client-remote-debugging.
- `body`: Lisp forms.

#### Values
- `body-values`: The values returned by `body`.

#### Description
The macro `with-remote-debugging-connection` dynamically binds an enabling switch controlling which remote debugging connection will be used by the remote debugging interface (entering the debugger, `start-remote-listener`, `remote-inspect`) on the client side while evaluating the forms in `body` as an implicit `progn`.

See `set-remote-debugging-connection` for details of how `connection` is interpreted.

Inside the dynamic extent of `body`, calls to `set-remote-debugging-connection` affect the switch only until the end of the `with-remote-debugging-connection` form.

#### Notes
In typical usage, you will not need to use `with-remote-debugging-connection`.

See also `set-remote-debugging-connection`
**with-remote-debugging-spec**  

*Macro*

**Summary**
Client side: Tell LispWorks how to open a connection for remote debugging on the client side within a dynamic extent.

**Package**
dbg

**Signature**
with-remote-debugging-spec (host &key port log-stream  
failure-function timeout open-callback) &body body => body-values

**Arguments**
- **host**
  A string specifying the IDE side hostname, or `nil`.
- **port**
  An integer.
- **log-stream**
  An output stream or `nil`.
- **failure-function**
  `nil` or a function of two arguments: Host and Port.
- **timeout**
  A non-negative `real` or `nil`.
- **open-callback**
  `nil` or a function that takes one argument, a newly opened connection.
- **body**
  Lisp forms.

**Values**
- **body-values**
  The values returned by `body`.

**Description**
The macro `with-remote-debugging-spec` establishes a dynamic extent of connection specification, calls `configure-remote-debugging-spec` passing it `host`, any supplied key-words (`port`, `log-stream`, `failure-function`, `timeout`, `open-callback`) and also `:setup-default nil :enable nil`. `body` is evaluated as an implicit `progn` in this dynamic extent. On exiting `with-remote-debugging-spec`, the connection specification reverts to what it was on entry.

The effect is to have a configured connection specification in the dynamic extent of `body` that is different from the global one, without having any effect on the global settings.
**with-remote-debugging-spec** returns the values returned by `body`.

See [configure-remote-debugging-spec](#) for the meaning of `host` and the other keywords.

See also [configure-remote-debugging-spec](#)
This chapter describes symbols available in the DSPEC package.

The dspec system is discussed in detail in Chapter 7, “Dspects: Tools for Handling Definitions”.

**active-finders**

Summary: Controls how source finding operates.

Package: dspec

Initial value: (:internal)

Description: The variable *active-finders* controls how the functions `find-name-locations` and `find-dspec-locations` operate. This in turn controls source the finding commands in the LispWorks IDE. You can switch between different sources of location information by setting this variable.

The legal values for the elements of *active-finders* are:
:internal  The internal database of definitions performed in this image.

tags  Prompt for a tags file, when first used.

pathname  Either a tags file or a tags database.

A tags database is a fast file generated by `save-tags-database`.

The order of this list determines the order that the results from the finders are combined in — you would usually want `:internal` to be the first item on this list, as it contains the up-to-date information about the state of the image. More than one pathname is allowed.

Notes  The value of `*active-finders*` is affected by editor commands such as Rotate Active Finders and Visit Tags File.

See also  `discard-source-info`  `find-dspec-locations`  `find-name-locations`  `save-tags-database`

---

**at-location**  

Macro

**Summary**  
Tells the dspec system of the source location.

**Package**  
dspec

**Signature**  
`at-location (location) &body body => result`

**Arguments**  
`location`  
A pathname or a keyword.

`body`  
Forms, including defining forms.

**Values**  
`result`  
The result of `body`. 
Description

The macro **at-location** informs the dspec system that the source for definitions done during the execution of **body** are at the location **location**.

**location** is usually a pathname, for definitions occurring in a file or editor buffer with that pathname.

Other locations are reserved for internal use. These are:

An editor buffer Defined in an editor buffer with no pathname.

:listener Interactively defined.

:unknown Defined without dspec information being recorded.

:implicit An aggregate defined by the existence of a part.

(:inside dspec loc)

A subform of dspec at location loc.

---

**canonicalize-dspec**

Function

Summary

Returns the canonical form for a dspec.

Package **dspec**

Signature

`canonicalize-dspec dspec => canonical-dspec`

Arguments

`dspec` A dspec.

Values

`canonical-dspec` A canonical dspec.

Description

The function **canonicalize-dspec** checks that **dspec** is syntactically correct and returns its canonical form if **dspec** is valid. Otherwise **canonicalize-dspec** returns **nil**.

**canonicalize-dspec** expands dspec aliases
Example

```
CL-USER 12 > (dspec:canonicalize-dspec 'foo)
(FUNCTION FOO)

CL-USER 13 > (dspec:canonicalize-dspec '(defmethod bar
(list t)))
(METHOD BAR (LIST T))
```

See also

```
define-dspec-alias
```

---

**def**

**Macro**

**Summary**

Informs the system of a name for a definition.

**Package**

dspec

**Signature**

def dspec &body body => result

**Arguments**

dspec A dspec.

body Lisp forms, evaluated as an implicit `progn`.

**Values**

result The result of body.

**Description**

The macro `def` informs the system that any definitions within `body` should be recorded as being within the dspec `dspec`. This means that when something attempts to locate such a definition, it should look for a definition named `dspec`.

Use `def` to wrap a group of definitions so that source location for one of the group causes the LispWorks Editor to look for the dspec in the `def` instead. Typically you will also need a `define-form-parser` definition for the macro that expands into the `def`.

`dspec` can be non-canonical.

You can also use `def` to provide a dspec for a definition that has its own class that has been defined with `define-dspec-`
class. In this case, you arrange to call record-definition with the same dspec as in the example below.

It is also possible to mix these cases, recording a dspec and also grouping inner definitions. For example defstruct does this, recording itself and also grouping definitions such as the constructor function.

In all cases, to make source location work in the LispWorks editor you typically also need a def-form-parser definition for the macro that expands into the def.

Example

```
(defmacro define-wibble (x y)
  `(dspec:def (define-wibble ,x)
    (set-wibble-definition ',x ',y (dspec:location))))

(defun set-wibble-definition (x y loc)
  (when (record-definition `(define-wibble ,x) loc)
    ;; defining code here
  ))
```

See also location

define-dspec-alias

Macro

Summary Inform the dspec system that a definer expands into another definer.

Package dspec

Signature define-dspec-alias name lambda-list &body body

Arguments name A symbol naming a definer.
lambda-list A list representing the parameters of a name dspec.
body Forms evaluated to yield a dspec.
The macro `define-dspec-alias` works rather like `deftype`. Dspecs whose `car` is `name` should have parameters that match `lambda-list`. They will be canonicalized into the dspec returned by `body`.

`define-dspec-alias` is useful when you add a new way of making existing definitions with a new defining form that expands into a system-provided defining form. The dspec system should consider the new and system-provided definers as variant forms of the same dspec class. `define-dspec-alias` is used to convert one of them to the other during canonicalization by `canonicalize-dspec`.

Example

`defparameter` is pre-defined as an alias for `defvar`.

See also `canonicalize-dspec`
prettify

A function to return a prettier form of a
dspec of the class.

definedp

A function to decide if a dspec of the class
currently has a definition.

object-dspec

A function to return the dspec from an
object if it was defined by the class.

defined-parts

A function to return all the currently defined
parts in the class for a given a primary-
name.

aggregate-class

The aggregate dspec class for a part dspec.

Description

The macro define-dspec-class defines a dspec class, pro-
viding handlers for definitions in that dspec class.

define-dspec-class defines name as a dspec class, inherit-
ing from the dspec class superspace. superspace should be nil
to define a new top-level dspec class.

documentation should be a string documenting the dspec
class. For example "My Objects".

After evaluating a define-dspec-class form, name can be
used by defining forms to record locations of definitions of
that dspec class name by calling record-definition.

All of the remaining arguments described below can be omit-
ted if not needed. The most important arguments for the
LispWorks IDE are definedp and undefiner.

If undefiner is supplied, its value must be a function of one
argument. When LispWorks wants to remove a definition, it
will call the function with a canonical dspec of class name.
The function should return a form that removes the current
definition of that dspec. For example, the undefining form for
package dspecs might be delete-package. If undefiner is
omitted, then definitions of this class cannot be undefined.

If canonicalize is supplied, its value must be a function of one
argument. The function will be called by canonicalize-
dspec for a dspec of the given class. The value returned by
the canonicalize function must be a fully canonical dspec of
the given class. A typical use for the canonicalize function
would be to remove extra options from the dspec which are
not required to make the dspec unique. The canonicalize
function should return nil for malformed dspecs and should
take care not to signal an error. The default canonicalize func-
tion returns the dspec if it has the form

\((\text{name symbol})\)

If prettify is supplied, its value must be a function of one argu-
ment. When LispWorks wants to print a dspec, for example
in an error message, it will call the prettify function for the
class of the dspec. The argument will be the canonical dspec
and the function should return a dspec which is considered
"prettier" for a user to see. The default prettify function
returns the dspec unchanged.

If definedp is supplied, its value must be function of one argu-
ment. When LispWorks wants to discover if a given dspec is
defined, it calls the function with the \text{dspec-primary-name}
of the dspec. The definedp function should return true if the
primary name is defined in this dspec class and nil other-
wise. The default definedp function always returns nil.

If object-dspec is supplied, its value must be a function of one
argument. When LispWorks wants to find the dspec that cre-
ated a given object (for example a package object created by a
defpackage form), it calls the object-dspec functions in all
dspec classes. The function should return a dspec for the
object if that object was defined by the dspec class or nil oth-
ewise. For example, the object-dspec function for package
dspecs might be:

```lisp
#'(lambda (obj)
   (and (packagep obj)
     `(package ,{package-name obj})))
```
The `object-dspec` function is used by the menu command **Find Source** in the LispWorks IDE Inspector tool to find where the current object was defined.

If `defined-parts` is supplied, its value must be a function of one argument. When LispWorks wants to find all the definitions that are parts of a given aggregate `dspec` class, it calls the `defined-parts` functions with the `dspec-primary-name` of the `dspec` in each class that aggregates with it. The function should return a list of `dspec`s which are defined parts of the primary name in the class `name`. If `defined-parts` is supplied, `aggregate-class` must also be supplied.

If `aggregate-class` is supplied and non-nil, its value must be a symbol naming a `dspec` class that is the aggregate class of the parts defined by `name` `dspec`s. For example, the aggregate class of `method` is `defgeneric` because methods are the defined parts of a particular generic function. If `aggregate-class` is supplied, the `defined-parts` must also be supplied. If `aggregate-class` is `nil` then `name` is not a part class.

To make `cl:documentation` work for your `dspec` class, add a suitable method as described for `documentation`.

**Example**

See “Dspec classes” on page 77.

**See also**

canonicalize-dspec
def
dspec-primary-name
record-definition

**define-form-parser**

*Macro*

**Summary**

Establishes a parser for top level forms with the given definer.

**Package**

dspec
The macro `define-form-parser` defines a form parser for forms beginning with `definer`. `options` is a property list with the following keys allowed:

- **:parser**
  A parser function `parser-function`.

- **:alias**
  A dspec class or alias `alias`.

- **:anonymous**
  A boolean.

The parser function defined is named by `parser-function`. If the `:parser` option is omitted then the name defaults to a symbol in the current package whose symbol name is the symbol name of `definer` with "-FORM-PARSER" appended.

If `parameters` and `body` are given, then `parser-function` is defined as a global function that is expected to return a dspec for the defining form or `nil` if this is not possible. Within `body`, `definer` is bound to the `car` of the actual form being parsed. In simple cases, this is just `definer`, but if the form parser is used as in the `:alias` option of another form parser then the symbol will be bound to the `car` of that form instead.
The *params* are bound to subsequent subforms of the defining form. If *rest param-getter* is supplied, then it is bound to a function of no arguments that returns two values: the next subform if there is one and a boolean to indicate if a subform was found.

If *parameters* and *body* are omitted, then *parser-function* is expected to be a form parser defined by a different *define-form-parser* form, or you can specify as an alias a definer with an existing form parser via the value *alias* of the :alias key in options.

If the :anonymous option is non-nil then *definer* is not associated with the form parser. This is useful in conjunction with *parameters* and *body* for defining generic form parsers that can be used in other *define-form-parser* forms.

LispWorks contains pre-defined form parser functions for the Common Lisp definers defun, defmethod, defgeneric, defvar, defparameter, defconstant, defstruct, defclass, defmacro and deftype and for LispWorks definers such as fli:define-foreign-type and dspec:define-form-parser itself.

When a defining symbol *definer* has an associated form parser, this parser function is used by the source location commands such as **Expression > Find Source** in the LispWorks IDE. Having identified the file where the definition was recorded, LispWorks parses the top level forms in the file looking for the one which matches the definition spec. When found, this match is displayed.

**Example**

Define a parser for *def-foo* forms which have a single name as the second element in the form:

**(dspec:define-form-parser def-foo (name)
  `(,(def-foo ,name))**

Define a parser for *def-other-foo* forms which are like *def-foo* forms:
(dspec:define-form-parser
  (def-other-foo (:parser def-foo-form-parser)))

Define a parser for def-bar forms whose name is made from the second element of the form and any subsequent keywords:

(dspec:define-form-parser def-bar (name &rest details)
  `(,def-bar ,(name @(loop for detail = (funcall details)
       while (keywordp detail)
       collect detail)))))

Define a parser for forms which have another name as the second element in the form:

(dspec:define-form-parser (two-names (:anonymous t)) (name1 name2)
  `,(two-names ,name1 ,name2))

Define a new way to define CLOS methods, and tell the dspec system to treat them the same. Note the use of define-dspec-alias to inform the dspec system that my-defmethod is another way of naming defmethod dspecs:

(defmacro my-defmethod (name args &body body)
  `(defmethod ,name ,args ,@body))

(dspec:define-dspec-alias my-defmethod
  (name &rest args)
  `,(defmethod ,name ,@args))

(my-defmethod foo ((x number)) 42)

(dspec:define-form-parser
  (my-defmethod
   (:parser #.(dspec:get-form-parser 'defmethod))))

A simpler way to write the last form is:

(dspec:define-form-parser
  (my-defmethod
   (:alias defmethod)))
See also get-form-parser
parse-form-dspec

discard-source-info Function

Summary Clears the internal dspec database.

Package dspec

Signature discard-source-info => nil

Arguments None.

Values Returns nil.

Description The function discard-source-info removes all source location information from the internal dspec database.

Example To build my-image which does not contain source locations for the definitions loaded, but retaining a tags database of those definitions:

(load-all-patches)
(load "my-code")
(dspec:save-tags-database
 (compile-file-pathname #P"my-tags-database"))
(dspec:discard-source-info)
(save-image "my-image")

See also save-tags-database

dspec-class Function

Summary Returns the dspec class of a dspec.

Package dspec

The DSPEC Package

36  The DSPEC Package

Signature  
\textbf{dspec-class} \textit{dspec} => \textit{class}

Arguments  
\textit{dspec} A dspec.

Values  
\textit{class} A dspec class name.

Description  
The function \textbf{dspec-class} returns the dspec class name for \textit{dspec}.

Example  
\begin{verbatim}
CL-USER 14 > dspec:dspec-class 'foo
FUNCTION

CL-USER 15 > dspec:dspec-class '(defmacro foo)
DEFMACRO

CL-USER 16 > dspec:dspec-class '(defmethod foo)
DEFMETHOD
\end{verbatim}

See also  
dspec-name

\textbf{*dspec-classes*}  \hspace{1cm} Variable

Summary  
Lists all the dspec classes.

Package  
dspec

Signature  
\textbf{*dspec-classes*}

Description  
The variable \textbf{*dspec-classes*} contains a list of the names of all the dspec classes.

\textbf{dspec-defined-p}  \hspace{1cm} Function

Summary  
The predicate for whether a dspec has a definition.

Package  
dspec
**Signature**

dspec-defined-p  dspec => definedp

**Arguments**

dspec  A dspec.

**Values**

definedp  The canonical form of dspec if dspec is defined, or nil otherwise.

**Description**

The function dspec-defined-p determines whether the dspec dspec has a definition. If so, it returns the canonical form of dspec.

If dspec has no definitions, dspec-defined-p returns nil.

**Example**

CL-USER 23 > (dspec:dspec-defined-p '(function list))
(DEFUN LIST)

**dspec-definition-locations**

**Function**

**Summary**

Returns the locations of the known definitions.

**Package**

dspec

**Signature**

dspec-definition-locations  dspec => locations

**Arguments**

dspec  A dspec.

**Values**

locations  A list of pairs (recorded-dspec location).

**Description**

The function dspec-definition-locations returns the locations of the definitions recorded for the dspec dspec.

For each known definition recorded-dspec names the definition that defined dspec in location, and location is a pathname or keyword as described in at-location.

Note that non-file locations, such as :unknown, can occur in the list. The locations in locations are all basic locations: that is, there are no (:inside ...) locations.
If `dspec` is a local `dspec`, the parent function is located.

Example

```lisp
CL-USER 6 > (dspec:dspec-definition-locations
  '(defun foo-bar))
(((DEFSTRUCT FOO) #P"C:/temp/hack.lisp")
```

See also `name-definition-locations`

---

**dspec-equal**

*Function*

**Summary**
Tests two `dspecs` for equality as `dspecs`.

**Package**

`dspec`

**Signature**

```
dsprep-equal dspec1 dspec2 => result
```

**Arguments**

`dspec1, dspec2` Dspecs.

**Values**

`result` A boolean.

**Description**

The function `dspec-equal` compares `dspec1` and `dspec2` for equality as `dspecs`.

Both arguments are canonicalized before the comparison.

Dspecs in different subclasses of the same namespace are `dspec-equal` if their names match.

Unknown `dspecs` are compared simply by `equal`.

**Example**

```lisp
CL-USER 44 > (dspec:dspec-equal
  '(deftype foo)
  '(defclass foo))
T
```

---

**dspec-name**

*Function*

**Summary**
Extracts the name from a canonical `dspec`.
**dspec-name**

*Function*

**Summary**

Extracts the name from the canonical dspec.

**Package**

dspec

**Signature**

dspec-name dspec => name

**Arguments**

dspec A canonical dspec.

**Values**

name A dspec name.

**Description**

The function `dspec-name` extracts the name from the canonical dspec `dspec`.

Note that for part classes this is a list starting with the primary name.

If `dspec` is not canonicalized, `dspec-name` signals an error.

**See also**

dspec-class

---

**dspec-primary-name**

*Function*

**Summary**

Extracts the primary name from a canonical dspec.

**Package**

dspec

**Signature**

dspec-primary-name dspec => name

**Arguments**

dspec A canonical dspec.

**Values**

name A dspec name.

**Description**

The function `dspec-primary-name` extracts the primary name from the canonical dspec `dspec`.

Note that for part classes this is the name of the aggregate definition, for example for methods it returns the name of the generic function.

**See also**

dspec-class
**dspec-progenitor**  
*Function*

**Summary**

Returns the ultimate parent of a `subfunction` dspec.

**Signature**

```lisp
 dspec-progenitor  dspec => result
```

**Package**

dspec

**Arguments**

- `dspec`  
  A dspec.

**Values**

- `result`  
  A dspec.

**Description**

The function `dspec-progenitor` returns a dspec `result` which is the ultimate parent of a `subfunction` dspec argument `dspec`.

If the argument `dspec` is not a local dspec, it is simply returned.

Note that `result` is not necessarily a canonical dspec.

**Example**

```lisp
(dspec-progenitor
 '(subfunction 1 (subfunction (flet a) (defun foo))))
=>
(defun foo)
```

**See also**

- `local-dspec-p`

---

**dspec-subclass-p**  
*Function*

**Summary**

Tests whether one dspec class is a subclass of another.

**Package**

dspec

**Signature**

```lisp
 dspec-subclass-p  class1 class2 => result
```

**Arguments**

- `class1, class2`  
  Symbols naming dspec classes.
Values

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A boolean.</td>
</tr>
</tbody>
</table>

Description

The function `dspec-subclass-p` determines whether the dspec class denoted by `class1` is a subclass of that denoted by `class2`.

Example

```
CL-USER 55 > (dspec:dspec-subclass-p 'defmacro 'type)  
NIL
CL-USER 56 > (dspec:dspec-subclass-p 'defmacro 'function)  
T
```

dspec-undefiner

Function

Summary

Returns an undefining expression for a dspec.

Package

dsproc

Signature

dspec-undefiner dspec => form

Arguments

dspec  
A dspec.

Values

<table>
<thead>
<tr>
<th>form</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Lisp form.</td>
</tr>
</tbody>
</table>

Description

The function `dspec-undefiner` returns a form which would undefine dspec, whether or not dspec is currently defined. If no such form can be constructed, nil is returned.

Example

```
CL-USER 66 > (dspec:dspec-undefiner '(defun foo))  
(PROGN (FMAKUNBOUND (QUOTE FOO)) (SETF (DOCUMENTATION (QUOTE FOO) (QUOTE FUNCTION)) NIL))
```

find-dspec-locations

Function

Summary

Returns the locations of the definitions of a dspec.
The function `find-dspec-locations` returns the locations of the relevant definitions for the dspec `dspec`.

For each known definition `recorded-dspec` names the definition that defined `dspec` in `location`, and `location` is a pathname or keyword as described in `at-location`.

If `dspec` is a local dspec, the parent function is located.

The location information is collected from all finders on `*active-finders*`, that is, the relevant definitions are those known to at least one of these finders.

If two or more finders return the same pair `(recorded-dspec location)`, as compared by `dspec-equal` and location equality, then only the first occurrence of the pair (in the order of `*active-finders*`) appears in `locations`.

See also:
- `*active-finders*
- dspec-definition-locations
- dspec-equal

---

**find-name-locations**

*Function*

**Summary**

Returns the locations of the definitions of a name.

**Package**

dspec

**Signature**

`find-name-locations classes name => locations`
Arguments  

`classes`  A list of dspec class names.

`name`  A name.

Values  

`locations`  A list of pairs (recorded-dspec location).

Description  
The function `find-name-locations` returns the locations of the relevant definitions for `name` in the classes listed in `classes`.

For each known definition `recorded-dspec` names the definition that defined `name` in `location`, and `location` is a pathname or keyword as described in `at-location`.

The location information is collected from all finders on `*active-finders*`, that is, the relevant definitions are those known to at least one of these finders.

If two or more finders return the same pair (recorded-dspec location), as compared by `dspec-equal` and location equality, then only the first occurrence of the pair (in the order of `*active-finders*`) appears in `locations`.

See also  

`*active-finders*`  
`name-definition-locations`  
`dspec-equal`  

---

**get-form-parser**  

*Function*

**Summary**  

Returns the form parser associated with a definer.

**Package**  

dspec

**Signature**  

`get-form-parser definer => parser`

**Arguments**  

`definer`  A symbol naming a definer.

**Values**  

`parser`  A form parser function, or `nil`.  

---
The function `get-form-parser` returns a form parser function if there is one associated with `definer`.

This is the case for predefined definers and for those for which you have established a form parser using `define-form-parser`.

If there is no associated form parser, `nil` is returned.

Example

CL-USER 1 > dspec: get-form-parser 'defun
DSPEC::NAME-ONLY-FORM-PARSER

See also

`define-form-parser`
`parse-form-dspec`

---

**local-dspec-p**

**Function**

**Summary**

The predicate for local dspecs.

**Package**

dspec

**Signature**

`local-dspec-p dspec => localp`

**Arguments**

dspec A dspec.

**Values**

localp A boolean.

**Description**

The function `local-dspec-p` determines whether the dspec `dspec` is a local dspec.

Local dspecs name local definitions, such as local functions.

Currently a local dspec is a list whose `car` is `subfunction`.

See also

dspec-progenitor
### location

**Macro**

**Summary**
Returns the source location.

**Package**
dspec

**Signature**
`location => location`

**Values**
`location`  A pathname or a keyword.

**Description**
The macro `location` returns a location suitable for passing to `record-definition`. This is usually done via a separate defining function. You will need to use `location` only if you create your own ways of making definitions (and not if your definers call only system-provided definers).

**Example**
```
(defmacro define-wibble (x y)
  `(dspec:def (define-wibble ,x)
    (set-wibble-definition ,',x ',y (dspec:location)))))
```
```
(defun set-wibble-definition (x y loc)
  (when (record-definition `(define-wibble ,x) loc)
    ;; defining code here
    ))
```

**See also**
at-location
def
record-definition

### name-defined-dspecs

**Function**

**Summary**
Returns defined dspecs matching a name.

**Package**
dspec

**Signature**
`name-defined-dspecs classes name => dspecs`

**Arguments**
`classes`  A list of dspec class names.
name     A name.

Values    dspecs    A list of canonical dspecs.

Description The function name-defined-dspecs looks in each of the
dspec classes classes for definitions of name.
For each definition found (as if by dspec-defined-p), the
result dspecs contains the canonical dspec.

See also    dspec-defined-p

name-definition-locations     Function

Summary       Returns the locations of the known definitions.

Package       dspec

Signature     name-definition-locations classes name => locations

Arguments     classes    A list of dspec class names.
               name       A name.

Values        locations  A list of pairs (recorded-dspec location).

Description   The function name-definition-locations returns the loca-
tions of the definitions recorded for the name name in any of
the dspec classes in classes.
For each known definition recorded-dspec names the definition
that defined name in location, and location is a pathname or
keyword as described in at-location.

Notes         name-definition-locations does not use *active-find-
ers*.
Example: CL-USER 7 > (dspec:name-definition-locations
'(function) 'foo-bar)
(((DEFSTRUCT FOO) '#P"C:/temp/hack.lisp")

See also: dspec-definition-locations

name-only-form-parser

Function

Summary: A pre-defined form parser.

Package: dspec

Signature: name-only-form-parser top-level-form getter => dspec

Arguments:

- top-level-form: A top-level defining form.
- getter: The subform getter function.

Values: dspec: A dspec.

Description: The function name-only-form-parser is a predefined form parser for use with define-form-parser. The parser consumes one subform and returns it.

name-only-form-parser can be used for function definitions where the function name is an abbreviation for the full dspec. It is the predefined parser for defun, defmacro and defgeneric forms.

You can define it to be the parser for your defining forms using define-form-parser.

Example:
(defmacro my-definer (name &body body)
 `((defun ,name (x)
       ,@body)))
(dspect:define-form-parser
  (my-definer (:parser dspec:name-only-form-parser))))
See also define-form-parser

object-dspec

Function

Summary
Returns the dspec of an object.

Package
dspec

Signature
object-dspec object => dspec

Arguments
 object Any object.

Values
dspec A dspec or nil.

Description
The function object-dspec returns a dspec for object if there is one, or nil otherwise. When the result dspec is not nil, it is a dspec as described in “Forms of dspecs” on page 76.

An object has a dspec only when it represents the result of some definition. The most useful cases are functions (of any kind) and methods, because their dspecs can be used to trace and advise them. Classes and packages also have dspecs.

Dspecs can also be useful for finding where the definition of the object originated, either by using dspec functions like find-dspec-locations, which returns the location, or using the editor command Find Source For Dspec, which edits them. The Common Lisp function ed also recognizes dspecs and goes to the source.

If object does not have a dspec then object-dspec returns nil.

See also trace
defadvice
find-dspec-locations
**parse-form-dspec**

*Function*

Summary  Parses the dspec from a defining form.

Package  dspec

Signature  `parse-form-dspec form => result`

Arguments  `form`  A form.

Values  `result`  A dspec or nil.

Description  The function `parse-form-dspec` invokes the defined form parser for `form` and returns the resulting dspec.

Example  
```
(parse-form-dspec '(def-foo my-foo (arg) (foo-it arg)))
=>
(def-foo my-foo)
```

See also  `define-form-parser`
`get-form-parser`

---

**record-definition**

*Function*

Summary  Checks for existing definitions and records a new definition.

Package  dspec

Signature  `record-definition dspec location &key check-redefinition-p => result`

Arguments  `dspec`  A dspec.

`location`  A pathname or keyword.

`check-redefinition-p`  A boolean.
Values

result A generalized boolean.

Description

The function \texttt{record-definition} tells the system that \texttt{dspec} is defined at \texttt{location}.

The system-provided definer macros call the function \texttt{record-definition} with the current location.

\texttt{location} should be a pathname or keyword as returned by \texttt{location}.

When \texttt{check-redefinition-p} is true, it checks for existing definitions and reports these according to the value of \texttt{*redefinition-action*}. The default value of \texttt{check-redefinition-p} is \texttt{t}.

If the definition is made, then \texttt{result} is true. If the definition is not made then \texttt{result} is \texttt{nil}. This can happen if you choose the "Don't redefine ..." restart at a redefinition error.

Notes

You should not usually call \texttt{record-definition}, since all the system-provided definers call it. However, for new classes of definition which you add with \texttt{define-dspec-class}, you should call \texttt{record-definition} for dspecs in their new classes.

Compatibility notes

\texttt{record-definition} was documented in the \texttt{lispworks} package in LispWorks 4.3 and earlier. Although it is currently still available there, this may change in future releases and you should now reference it via the \texttt{dspec} package.

See also

\texttt{define-dspec-class}

\texttt{*redefinition-action*}

\texttt{location}

"Recording definitions and redefinition checking" on page 87

\texttt{*record-source-files*}

Variable

Summary

Controls whether the locations of definitions are recorded.
<table>
<thead>
<tr>
<th>Package</th>
<th>dspec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial value</td>
<td>t</td>
</tr>
<tr>
<td>Description</td>
<td>The variable <em>record-source-files</em> controls whether locations of definitions are recorded in the internal tags database.</td>
</tr>
<tr>
<td>Compatibility notes</td>
<td><em>record-source-files</em> was documented in the lisp-works package in LispWorks 4.3 and earlier. Although it is currently still available there, this may change in future releases and you should now reference it via the dspec package.</td>
</tr>
<tr>
<td>See also</td>
<td><em>active-finders</em></td>
</tr>
</tbody>
</table>

### *redefinition-action* Variable

<table>
<thead>
<tr>
<th>Summary</th>
<th>Specifies the action on some redefinitions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>dspec</td>
</tr>
<tr>
<td>Initial value</td>
<td>:warn</td>
</tr>
<tr>
<td>Description</td>
<td><em>redefinition-action</em> controls messages about redefinitions seen by the source location system. If <em>redefinition-action</em> is set to :warn then you are warned. If it is set to :quiet or nil, the redefinition is done quietly. If, however, it is set to :error, then LispWorks signals an error. These messages are triggered by defining forms provided, but they could also be from any call to record-definition.</td>
</tr>
<tr>
<td>Notes</td>
<td><em>redefinition-action</em> does not affect the behavior of cl:defstruct.</td>
</tr>
</tbody>
</table>
Compatibility notes

*redefinition-action* is documented in the lispworks package in LispWorks 4.3 and earlier. It is still currently still available there but this may change in future releases and you should now reference it via the dspec package.

See also

*handle-warn-on-redefinition*
record-definition

replacement-source-form

Macro

Summary Allows source location to work when a form is copied by a macro.

Package dspec

Signature replacement-source-form original-form new-form => new-form-value

Arguments

original-form A Lisp form.

new-form A Lisp form.

Values

new-form-value A Lisp object.

Description A call to replacement-source-form can be used to allow the debugger and stepper to identify that original-form has been replaced by new-form in a macro expansion. Forms in a macro expansion that are eq to forms in the arguments to the macro will be identified automatically, but some macros (such as iterate) need to generate new forms that are equivalent to the original forms and wrapping them with replacement-source-form allows them to be identified too.

original-form should be a form that occurred in the arguments to the macro and does not otherwise occur in the expansion of the macro. new-form should be a form that was created by the macro.
The value of new-form, new-form-value, is returned when the replacement-source-form form is evaluated.

Examples

Without the dspec:replacement-source-form, the calls to pprint would be unknown to the debugger and stepper because the forms do not occur in the original source code:

```lisp
(defun pprint-for-print (&body forms)
  `(progn
     ,@(loop for form in forms
            collect
            (if (and (consp form)
                     (eq (car form) 'print))
                ,(dspec:replacement-source-form
                   ,form
                   (pprint ,(cdr form)))
               form))))
```

**save-tags-database**

*Function*

**Summary**
Saves the current internal dspec database to a given file.

**Package**
dspec

**Signature**
save-tags-database pathname => pathname

**Arguments**
pathname A filename.

**Values**
pathname The filename that was supplied.

**Description**
The function save-tags-database saves the current internal dspec database into the file given by pathname. The file can then be used in the variable *active-finders*.

**See also**
*active-finders*
discard-source-info
The function `single-form-form-parser` is a predefined form parser for use with `define-form-parser`. The parser consumes one subform and returns a dspec made from the defining form and the subform. This can be used in the common case where a defining form has a name that follows the defining macro and the dspec class is the same as the defining macro, for example `defclass`

`single-form-form-parser` is the predefined parser for `defvar`, `defparameter`, `defconstant`, `define-symbol-macro`, `define-compiler-macro`, `deftype`, `defsetf`, `define-setf-expander`, `defpackage`, `defclass`, `define-condition` and `define-method-combination` top level forms. It is also the parser for various LispWorks extensions such as `defsystm`.

You can define it to be the parser for your defining forms using `define-form-parser`.

See also: `define-form-parser`
Package  
dspec

Signature  
\textit{single-form-with-options-form-parser} \textit{top-level-form} \textit{getter} \text{=>} \textit{dspec}

Arguments  
\textit{top-level-form} \hspace{1em} \text{A top level defining form.}
\textit{getter} \hspace{1em} \text{The subform getter function.}

Values  
\textit{dspec} \hspace{1em} \text{A dspec.}

Description  
The function \textit{single-form-with-options-form-parser} is a predefined form parser for use with \textit{define-form-parser}. The parser consumes one subform and returns a dspec made from the defining form and either the first element of the subform if it is a cons or the subform itself otherwise. This can be used in the common case where a defining form has a name with options that follows the defining macro and the dspec class is the same as the defining macro, for example \textit{defstruct}.


You can define it to be the parser for your defining forms using \textit{define-form-parser}.

See also  
\textit{define-form-parser}

\texttt{traceable-dspec-p}  \hspace{1em} \textit{Function}

Summary  
Tests whether definition can be traced.

Package  
dspec
### The DSPEC Package

#### Signature
```
traceable-dspec-p  dspec => result
```

#### Arguments
- **dspec**: A dspec.

#### Values
- **result**: A generalized boolean.

#### Description
The function `traceable-dspec-p` determines whether the `dspec` `dspec` denotes a definition that can be traced using the Common Lisp macro `trace`.

`dspec` must not be a local dspec, and must be defined, according to `dspec-defined-p`. The result does not depend on whether `dspec` is currently traced.

#### Example
```
CL-USER 67 > (dspec:traceable-dspec-p '(subfunction foo bar))
NIL

CL-USER 68 > (dspec:traceable-dspec-p '(defun open))
OPEN
```

---

#### tracing-enabled-p

**Function**

#### Summary
Gets and sets the global tracing state.

#### Package
```
dspec
```

#### Signature
```
tracing-enabled-p => enabledp
(setf tracing-enabled-p) enabledp => enabledp
```

#### Values
- **enabledp**: A generalized boolean.

#### Description
The function `tracing-enabled-p` determines whether tracing (by the Common Lisp macro `trace`) is currently on. This is independent of whether any functions are currently traced.
The function `(setf tracing-enabled-p)` switches tracing on or off according to the value of `enabledp`. This does not affect the list of functions that are currently traced.

See also  
`trace`  
`tracing-state`

**tracing-state**  
*Function*

**Summary**  
Gets the current trace details.

**Package**  
dspec

**Signature**  
`tracing-state &optional dspec => state`

**Signature**  
`(setf tracing-state) state &optional dspec => state`

**Arguments**  
`dspec`  
A dspec.

**Values**  
`state`  
A list.

**Description**  
The function `tracing-state` returns a listing describing the current state of the tracing system. It shows the current tracing state for the dspec `dspec`, or for all traced definitions if `dspec` is not supplied.

The result `state` is a list each element of which is a list whose car is a dspec naming the traced definition and whose cdr is the additional trace options. Note that `tracing-state` returns more information than is returned by `trace`. It is useful for preserving a complex set of traces.

The function `(setf tracing-state)` sets the state of the tracing system. It changes the current tracing state for the dspec `dspec`, or for all traced definitions if `dspec` is not supplied.
(setf tracing-state) can be used to switch between different sets of traces. Note however that turning tracing on or off is better done using tracing-enabled-p.

See also

trace

tracing-enabled-p
The EXTERNAL-FORMAT Package

This chapter describes symbols available in the EXTERNAL-FORMAT package, along with some of the actual external formats (typically with keyword names).

Use of these symbols are discussed in Chapter 26, “Internationalization: characters, strings and encodings”.

:bmp
:bmp-native
:bmp-reversed

Summary
Implement reading and writing of 16-bit characters only (excluding supplementary characters).

Signature
:bmp &key use-replacement little-endian
:bmp-native &key use-replacement
:bmp-reversed &key use-replacement
Description

The external format :bmp and its variants implement reading and writing of 16-bit characters only (excluding supplementary characters).

:bmp-native and :bmp-reversed are the actual implementation formats. They implement reading and writing 16-bit characters with the native byte order (:bmp-native) or the reversed byte order (:bmp-reversed).

:bmp implements reading and writing 16-bit characters with control over the byte order. This format maps to either :bmp-native or :bmp-reversed as appropriate.

If little-endian is supplied, it determines the byte order. Otherwise, if it is used for opening a file, the system checks whether the file starts with the BOM (Byte Order Mark), and if so it uses it. Otherwise the native byte order is used. The system uses the required byte order and the native byte order of the computer it executes on to decide whether to use :bmp-native or :bmp-reversed.

When writing, these :bmp external formats signal an error when trying to write supplementary characters (code greater than #xffff).

:bmp cannot read surrogate code points. When encountering a surrogate code point it either signals an error (the default), or if use-replacement is non-nil, replaces it with the replacement character. When use-replacement is non-nil, these external formats never signal an error when reading.

Compatibility note:

These formats were new in LispWorks 7.0. In LispWorks 6.1 and earlier versions :unicode is the external format to read 16-bit characters. Other than the treatment of surrogate code points, :bmp now does what :unicode used to do.

See also

“16-bit External formats guide” on page 445
:unicode

**Summary**
Implements UTF-16 translation.

**Signature**
:unicode &key little-endian

**Description**
The external format :unicode implements UTF-16 translation, with default byte order the native one. :unicode is equivalent to (:utf-16 :little-endian little-endian) where the value of little-endian depends on the byte order of the native machine.

When opening a file with :external-format :unicode (without passing little-endian), the system checks for the existence of the BOM (Byte Order Mark) in the beginning of the file, and if there is a BOM uses it to determine the correct byte order. Otherwise, it uses the native byte order. There are no checks for a BOM in other situations.

**Notes**
:unicode differs from :utf-16 when little-endian is not supplied and there is no BOM, because :unicode uses the native endianness and :utf-16 uses big-endian. In all other circumstances :unicode is equivalent to :utf-16.

**Compatibility note**
In LispWorks 6.1 and earlier versions, :unicode reads only 16-bit characters, including character objects corresponding to surrogate code points. There is no exact match to that in LispWorks 7.0 and later, because there is no external format that reads surrogates. :bmp can be used to read 16-bit characters, either giving an error or using the replacement character for surrogate code points.

**See also**
"16-bit External formats guide" on page 445
:utf-16
:utf-16be
:utf-16le
:utf-16-native
:utf-16-reversed

**Summary**
Implement translations according to the UTF-16 standard of Unicode.

**Signature**

:utf-16 &key use-replacement little-endian
:utf-16be &key use-replacement
:utf-16le &key use-replacement
:utf-16-native &key use-replacement
:utf-16-reversed &key use-replacement

**Description**
The external format :utf-16 and variants implement translations according to the UTF-16 standard of Unicode.

The variants differ only in their treatment of byte order.

The parameter `use-replacement` is a boolean which defaults to `nil`. It controls what happens when reading encounters an illegal combination. Illegal combinations are either a leading surrogate (`#xd800` to `#xdbff`) not followed by a trailing surrogate (`#xdc00` to `#xdfff`), or a trailing surrogate not following a leading surrogate. By default, the input code signals an error of type `external-format-error`. If `use-replacement` is non-nil, the input code replaces the error byte or pair of bytes by the replacement character (`#xfffd`).

:utf-16-native and :utf-16-reversed implement UTF-16 in the native or the reverse of the byte order of the computer that they are executing on.

:utf-16be and :utf-16le implement the UTF-16BE and UTF-16LE standard format, that is UTF-16 big-endian and
UTF-16 little-endian. The system maps these to either of
:utf-16-native or :utf-16-reversed.

:utf-16 implements the UTF-16 standard. The byte order
defaults to big-endian byte order.

When opening a file with :external-format :utf-16
(without passing little-endian), the system checks for the exist-
ence of the BOM (Byte Order Mark) in the beginning of the
file, and if there is a BOM uses it to determine the right byte
order. Otherwise, it uses big-endian (:utf-16be). There are
no checks for a BOM in other situations.

Compatibility
notes
These formats were new in LispWorks 7.0.

In LispWorks 6.1 and earlier versions the :unicode external
format is the format to read 16-bit characters.

See also
“16-bit External formats guide” on page 445

:utf-32
:utf-32le
:utf-32be
:utf-32-native
:utf-32-reversed

Summary
Implement UTF-32 format, which means reading and writing
32-bit chunks as characters.

Signature
:utf-32 &key use-replacement little-endian
:utf-32le &key use-replacement
:utf-32be &key use-replacement
:utf-32-native &key use-replacement
:utf-32-reversed &key use-replacement

External Formats
The external format `:utf-32` and its variants implement UTF-32 format, which means reading and writing 32-bit chunks as characters.

`:utf-32-native` and `:utf-32-reversed` are the actual implementation formats. They implement UTF-32 with the native byte order (`:utf-32-native`) or the reversed byte order (`:utf-32-reversed`).

`:utf-32le` and `:utf-32be` implement UTF-32 with little-endian (`:utf-32le`) and big-endian (`:utf-32be`) byte order. The system maps them to one of `:utf-32-native` or `:utf-32-reversed` as appropriate.

`:utf-32` implements UTF-32 with control over the byte order. This format maps to one of `:utf-32-native` or `:utf-32-reversed` as appropriate. If `little-endian` is supplied, it determines the byte order. Otherwise, if it is used for opening a file, the system checks whether the file starts with the BOM (Byte Order Mark), and uses it if found. Otherwise the big-endian order is used. The system uses the required byte order and the native byte order of the computer it executes on to decide whether to use `:utf-32-native` or `:utf-32-reversed`.

If the `:utf-32` formats encounter a surrogate code point or a character code which is too large, they by default signal an error of type `external-format-error`. If `use-replacement` is non-nil, they replace the illegal input by the replacement character. When `use-replacement` is non-nil these formats never signal an error.

When writing, the `:utf-32` formats never signal an error.

These formats were new in LispWorks 7.0. In LispWorks 6.1 and earlier versions there is an undocumented external format `character` that works similarly to `:utf-32-native` in LispWorks 7.0 and later. This is now mapped to `:utf-32-native` to avoid errors in existing code, and should not be used in new code.
char-external-code

Function

Summary
Returns the code of a character in the specified character set.

Package
external-format

Signature
char-external-code char set => code

Arguments
char
The character whose code you wish to return.

set

Values
code
The code of char in the character set set. An integer.

Description
Returns the code of the character char in the coded character set specified by set, or nil, if there is no encoding. Note that a coded character set is not the same thing as an external format.

For the set parameter, the :jis-* codes are KUTEN indexes (from the 1990 version of these standards) encoded as

(+ (* 100 row) column)

:euc-jp is the complete two-byte format encoded as

(+ (* 256 first-byte) second-byte)

:sjis is Shift-JIS encoded in the same way. Strictly speaking, EUC and Shift-JIS are not coded character sets, but encodings
of the JIS sets, but the encoding is easily expressed as an integer, so the same interface to it is used.

See also  
find-external-char

**decode-external-string**  
*Function*

**Summary**  
Decodes a binary vector to make a string.

**Package**  
*external-format*

**Signature**  
`decode-external-string vector external-format &key start end => string`

**Arguments**  
vector A binary vector.

external-format An external format spec.

start, end Bounding index designators of vector.

**Values**  
string A string.

**Description**  
The function `decode-lisp-string` decodes the integers in the part of the vector `vector` bounded by `start` and `end` using encoding `external-format` to make a string `string`.

The element type of `vector` does not need to match the `external-format-foreign-type` of `external-format`.

**Compatibility notes**  
This function exists in LispWorks 5.0 but is not documented and does not take the :start and :end arguments. Also, it was inefficient prior to LispWorks 5.0.1.

See also  
encode-lisp-string
**encode-lisp-string**

*Function*

**Summary**
Converts a string to an encoded binary vector.

**Package**
external-format

**Signature**
`encode-lisp-string string external-format &key start end => vector`

**Arguments**
- `string` A string.
- `external-format` An external format spec.
- `start, end` Bounding index designators of `string`.

**Values**
`vector` A binary vector.

**Description**
The function `encode-lisp-string` converts the part of `string` bounded by `start` and `end` to a binary vector `vector` encoded in encoding `external-format`.

The element type of `vector` matches the `external-format-foreign-type` of `external-format`.

**Compatibility notes**
This function exists in LispWorks 5.0 but is not documented and does not take the :start and :end arguments. Also, it was inefficient prior to LispWorks 5.0.1.

**See also**
`decode-external-string`

**external-format-error**

*Condition*

**Summary**
The condition class `external-format-error` is the superclass of all errors relating to external formats.

**Package**
external-format

**Superclasses**
error
The class `external-format-error` provides a slot for the name of external format involved: this is the fully expanded form of the specification with all the parameters filled in. It is also useful for users who want to set up a handler for encoding errors.

**external-format-foreign-type**

*Function*

**Summary**
Returns a type specifier for the integers handled by a specified external format.

**Package**
`external-format`

**Signature**
```
external-format-foreign-type external-format => type-specifier
```

**Arguments**
`external-format` An external character format.

**Values**
`type-specifier` A type specifier describing the integer types handled by `external-format`.

**Description**
Takes the name of an external format, and returns a Lisp type specifier for the type of integers that the external format handles on the foreign side.

**See also**
`external-format-type`

**external-format-type**

*Function*

**Summary**
Returns a type specifier for the characters handled by a specified external format.

**Package**
`external-format`
Signature  
\texttt{external-format-type external-format => type-specifier}

Arguments  
\texttt{external-format}  
An external character format.

Values  
\texttt{type-specifier}  
A type specifier describing the character types handled by \texttt{external-format}.

Description  
Takes the name of an external format, and returns a type specifier for the type of characters that the external format handles on the Lisp side.

See also  
\texttt{external-format-foreign-type}

\textbf{find-external-char}  
\textit{Function}

Summary  
Returns the character of a given code in a specified character set.

Package  
\texttt{external-format}

Signature  
\texttt{find-external-char code set => char}

Arguments  
\texttt{code}  
A character code. This is an integer.
\texttt{set}  
A character set. Legal values for \texttt{set} are \texttt{:unicode}, \texttt{:latin-1}, \texttt{:ascii}, \texttt{:macos-roman}, \texttt{:jis-x-208}, \texttt{:jis-x-212}, \texttt{:euc-jp}, \texttt{:sjis}, \texttt{:koi8-r}, \texttt{:windows-cp936} and \texttt{:gbk}. Additionally, on Windows, \texttt{set} can be a valid Windows code page identifier.

Values  
\texttt{char}  
The character represented by \texttt{code}. If \texttt{code} is not a legal code in the specified set, the return value is undefined.

Description  
Returns the character that has the code \texttt{code} (an integer) in the coded character set specified by \texttt{set}, or \texttt{nil}, if that character is
not represented in the Lisp character set. Note that a coded character set is not the same thing as an external format.

For the set parameter, the :jis-* codes are KUTEN indexes (from the 1990 version of these standards) encoded as

\[ (+ (* 100 \text{row}) \text{column}) \]

:euc-jp is the complete two-byte format encoded as

\[ (+ (* 256 \text{first-byte}) \text{second-byte}) \]

:sjis is Shift-JIS encoded in the same way. Strictly speaking, EUC and Shift-JIS are not coded character sets, but encodings of the JIS sets, but the encoding is easily expressed as an integer, so the same interface to it is used.

See also char-external-code

valid-external-format-p

Function

Summary
Tests whether an external format spec is valid.

Package external-format

Signature valid-external-format-p ef-spec &optional env => result

Arguments
\begin{itemize}
  \item \texttt{ef-spec} An external format spec.
  \item \texttt{env} An environment across which the spec should apply.
\end{itemize}

Values result A boolean.

Description This predicate tests whether the external format spec given in \texttt{ef-spec} is valid (in the environment \texttt{env}). \texttt{result} is \texttt{t} if \texttt{ef-spec} is a valid spec, and \texttt{nil} otherwise.

Example
\begin{verbatim}
  (valid-external-format-p '(:Unicode :eol-style :lf))
\end{verbatim}
This chapter describes symbols available in the HCL package. This package is used by default. Its symbols are visible in the CL-USER package.

Various uses of the symbols documented here are discussed throughout this manual.

**Functions**

- **add-code-coverage-data**
- **reverse-subtract-code-coverage-data**
- **subtract-code-coverage-data**
- **destructive-add-code-coverage-data**
- **destructive-subtract-code-coverage-data**
- **destructive-reverse-subtract-code-coverage-data**

**Summary**
Add or subtract two code-coverage-data objects.

**Package**
hcl

**Signature**

- add-code-coverage-data $ccd1$ $ccd2$ name => new-ccd
- subtract-code-coverage-data $ccd1$ $ccd2$ name => new-ccd
reverse-subtract-code-coverage-data ccd1 ccd2 name => new-ccd
destructive-add-code-coverage-data ccd1 ccd2 => ccd1
destructive-subtract-code-coverage-data ccd1 ccd2 => ccd1
destructive-reverse-subtract-code-coverage-data ccd1 ccd2 => ccd1

Arguments

ccd1  A code-coverage-data object or (for the non-destructive functions only) t.
ccd2  A code-coverage-data object or t.
name  A Lisp object, normally a symbol or a string.

Values

new-ccd  A code-coverage-data object.

Description

Adding (subtracting) code coverage datas means adding (subtracting) all pairs of counters for the same piece of code from the two datas. When the data contains actual counters, adding (subtracting) really means adding (subtracting) the counter values, and reverse subtract means subtracting the first argument from the second. When the data contains only binary flags (that is, the code was compiled with counters = nil, see generate-code-coverage), addition is performed by doing logical OR, and subtraction by doing logical AND-NOT. Note that having counters is a property of each individual file, and a code-coverage-data object may have files that are compiled with either of these options.

If ccd1 or ccd2 has value t, this is interpreted as the internal code-coverage-data object.

These functions operate on each file in ccd1 (first argument), and for each of these file for which there is a match in ccd2, perform the operation on all the counters of this file. That is, they add (subtract) the counter from ccd2 to (from) the matching counter in ccd1. If there is no matching file in ccd2, the operation is done with 0 so the information from ccd1 is used unchanged.
For files which have matches in ccd2, the information must be based on the same binary file, otherwise these functions signal an error.

The functions `add-code-coverage-data`, `subtract-code-coverage-data` and `reverse-subtract-code-coverage-data` all produce a new `code-coverage-data` object (with name `name`) which is the result of the operation. The functions `destructive-add-code-coverage-data`, `destructive-subtract-code-coverage-data` and `destructive-reverse-subtract-code-coverage-data` all overwrite `ccd1` with the result and return it.

For all these functions the result is a `code-coverage-data` object with information for each file for which there is information in `ccd1`, combined with the counters from `ccd2` for files with a match. Files in `ccd2` for which there is no match in `ccd1` are ignored.

**Notes**

For `reverse-subtract-code-coverage-data` and `destructive-reverse-subtract-code-coverage-data` the result for files with no match may be considered inconsistent, because negation their counters may be more consistent.

**See also** Chapter 10, “Code Coverage”

### add-special-free-action

**Summary**

Adds a function to perform a special action during garbage collection.

**Package**

hcl

**Signature**

`add-special-free-action function => function-list`

**Arguments**

`function` A function designator for a function of one argument.
Values

**function-list**

A list of the functions currently called to perform special actions, including the one just added.

Description

When some objects are garbage collected, you may require a “special action” to be performed as well. **add-special-free-action** adds the function *function* to perform the special action. Note that the function is applied to all objects flagged for special-free-action, so the function *function* should check for the object’s type, so that it only affects relevant objects. Also, it should be fast when called with other objects.

The functions **flag-special-free-action** and **flag-not-special-free-action** flag and unflag objects for action.

When *function* is called, the object is still alive but is no longer flagged for special free action. Normally, the object will be collected on the next garbage collection cycle, but you can also store it somewhere which will prevent this. It may even be passed to **flag-special-free-action** again.

Example

```lisp
(defun free-my-app (object)
  (when (my-app-p object)
    (free-some-external-resources object)))

(add-special-free-action 'free-my-app)
```

See also

- **remove-special-free-action**
- **flag-special-free-action**
- **flag-not-special-free-action**

---

**add-symbol-profiler**

**Function**

Summary

Deprecated. Adds a symbol to the list of profiled symbols.

Package

**hcl**

Signature

```lisp
add-symbol-profiler symbol => nil
```
Arguments   

symbol  

A symbol to be added to the *profile-symbol-list*.

Values  

Returns nil.

Description  

The function add-symbol-profiler is deprecated. It adds a symbol to the list of profiled symbols.

See also  

remove-symbol-profiler

**allocation-in-gen-num**  

*Macro*

Summary  

Allocates objects from a specified generation within the scope of evaluating a number of forms in 32-bit LispWorks.

Package  

hcl

Signature  

allocation-in-gen-num gen-num &body body => result

Arguments  

gen-num  

An integer, which if out of range for a valid generation number is rounded either to the youngest or oldest generation.

If gen-num is negative, the specified generation is: the highest generation number + 1 – gen-num, so that an argument of –1 specifies the highest generation number.

body  

The forms to be evaluated while the allocation generation has been temporarily set to gen-num.

Values  

result  

The result of evaluating body.

Description  

Allocates objects from a specified generation during the extent of the evaluation of the body forms.
Normally objects are allocated from the first (youngest) generation, which assumes that they are short-lived. The memory allocator and garbage collector perform better if allocation of large numbers of non-ephemeral objects is done explicitly into a generation other than the youngest.

Notes

`allocation-in-gen-num` is implemented only in 32-bit LispWorks. In 64-bit implementations, use `apply-with-allocation-in-gen-num` or the `:allocation` argument to `make-array` instead.

Examples

```lisp
(allocation-in-gen-num 1
  (setq tab (make-hash-table :size 1200 :test 'eq)
    arr (make-array 20)))
```

See also

`apply-with-allocation-in-gen-num`
`make-array`
`set-default-generation`
`get-default-generation`
`*symbol-alloc-gen-num*`

“Memory Management in 32-bit LispWorks” on page 135

---

### analysing-special-variables-usage

**Function**

**Summary**

Prints an analysis of proclaimed symbols seen during compilation, as an aid to improving declarations.

**Package**

`hcl`

**Signature**

`analysing-special-variables-usage (&key all default maybe-globals maybe-dynamics unused only-bound wrong-global inconsistent stream) &body body => results`

**Arguments**

- `all` A boolean.
- `default` A boolean.
Values

maybe-globals  A boolean.
maybe-dynamics A boolean.
unused        A boolean.
only-bound    A boolean.
wrong-global  A boolean.
inconsistent  A boolean.
stream        t or an output stream.
body          Lisp code that calls the compiler.
results       The results of running body.

Description

The macro **analyzing-special-variables-usage** executes the code in body, which needs to call the compiler, typically many times (compiling a whole system, for example). When body exits, it prints a simple analysis of symbols that were proclaimed and how they were proclaimed, in a way that is intended to be helpful in improving declarations. For a full explanation of how you might add or alter declarations, see “Usage of special variables” on page 113.

The analysis is based solely on what the compiler sees, ignoring what is already in the image. It also ignores inline declarations.

Only symbols for which the compiler sees a special proclamation are reported (including cl:defvar, cl:defparameter, defglobal-parameter and defglobal-variable, but not cl:defconstant).

**all** and **default** are convenience arguments to control groups of the other keyword arguments, which are all boolean flags. The default value of all is nil. all provides the default value of maybe-globals and maybe-dynamics. The default value of default is t. default provides the default value of unused, only-bound, wrong-global and inconsistent.
stream determines where the analysis goes, and is interpreted as if by \texttt{cl:format}. It does not affect any of the I/O in \textit{body}. The default value of \textit{stream} is \texttt{t}, meaning standard output.

\textit{inconsistent} controls whether to print symbols where the declaration and usage is inconsistent. Inconsistencies include:

1. Accessing or binding the symbol before the proclamation.
2. Multiple declarations which are different (for example, change from \texttt{hcl:special-dynamic} to \texttt{cl:special})

The \textit{inconsistent} messages are the most useful. A well written program should not produce any such message.

\textit{unused} controls whether to report symbols that are proclaimed special but are otherwise not used. For this option to be really useful, \textit{body} needs to force compile many source files.

Since such unused variables do not affect the code, \textit{unused} is normally useful only for finding and eliminating dead declarations, but it can also flag situations when the wrong variable is used (if the variable that is supposed to be used is not used elsewhere)

\textit{only-bound} controls whether to report symbols that have been seen bound, but whose value has not been read. The comments about \textit{unused} also apply to \textit{only-bound}.

\textit{wrong-global} controls whether to print symbols that are bound but are also proclaimed \texttt{hcl:special-global}. If the proclamation preceded the binding, the compiler will signal a \texttt{compiler-error}.

\textit{maybe-globals} controls whether to report symbols that were not seen bound. If these symbols are really never bound, they can be proclaimed global by defining them with \texttt{defglobal-parameter} and \texttt{defglobal-variable}, or proclaimed \texttt{hcl:special-global}, both for speed and also to prevent them getting bound by mistake.
It is quite useful to force compile a program each now and then with `maybe-globals` true, then check through the report and proclaim global all those symbols that can be proclaimed global.

`maybe-dynamics` controls whether to report symbols that have been seen bound, and are proclaimed special, but not `hcl:special-dynamic` or `hcl:special-global`. Some of these may be proclaimed `hcl:special-dynamic`.

The report that is generated is grouped according to the file in which a proclamation was found. If a variable was proclaimed in multiple files, it will appear multiple times in the output. Within each file the output is grouped according to what is reported.

For the keyword arguments except `inconsistent`, the symbols are simply listed. For the `inconsistent` report, it outputs several lines for each symbol. Each line starts with one of the symbols `cl:special, hcl:special-global, hcl:special-dynamic, hcl:special-fast-access` (these four signify a proclamation), `:bound` or `:accessed` (these two indicate the usage). It is followed by the pathname of the file in which this one found. Only occurrences which give rise to inconsistency are listed.

**Notes**

The report about `inconsistent` usage is almost always useful. `unused` and `only-bound` are mostly useful when `body` force compiles many files, though they have limited utility in partial compilation too. `maybe-globals` and `maybe-dynamics` need full compilation to be really useful. Of the latter `maybe-globals` is the more useful.

**See also**

- `declare`
- `defglobal-parameter`
- `defglobal-variable`
<table>
<thead>
<tr>
<th>android-build-value</th>
<th>Function</th>
</tr>
</thead>
</table>

**Summary**
Android only: Returns the value of a field in the `android.os.Build` Java class.

**Package**
hcl

**Signature**
`android-build-value &optional name => result`

**Arguments**

<table>
<thead>
<tr>
<th>name</th>
<th>nil or a string.</th>
</tr>
</thead>
</table>

**Values**

<table>
<thead>
<tr>
<th>result</th>
<th>A string.</th>
</tr>
</thead>
</table>

**Description**
The function `android-build-value` is defined only in Lisp-Works for Android Runtime images, and can be used only when running on Android. It returns values of fields from the `android.os.Build` Java class.

If `name` is non-nil, it must a string naming a field in the `android.os.Build` Java class. `result` in this case is the value of this field.

If `name` is `nil` (the default), then `android-build-value` returns a string containing the names and values of most of the fields in the `android.os.Build` class. For each field, `result` contains a substring of the form:

"<field-name> : <field-value><newline>"

`result` is the concatenation of all of these substrings. The fields that are looked up are:
android-funcall-in-main-thread

android-funcall-in-main-thread-list

Functions

Summary Call a function on the Android main (GUI) thread.

Package hcl

Signature

android-funcall-in-main-thread func &rest args

Signature

android-funcall-in-main-thread-list func-and-args

Arguments

func A function designator.

args Arguments for func.

func-and-args A cons (func . args).

Description The functions android-funcall-in-main-thread and android-funcall-in-main-thread-list arrange for the
function func to be applied to the args on the Android main thread (which is the GUI thread too). `android-funcall-in-main-thread` actually does it by consing func and args and calling `android-funcall-in-main-thread-list` with the result. `android-funcall-in-main-thread-list` is the "primitive" interface.

The invocation of the function is done by the event loop of the GUI thread, so it is synchronous with respect to processing events, in other words it will not happen in the middle of processing an event.

These functions should be used when func does something that needs to run on the main thread, most commonly operations that interact with GUI elements.

To allow for testing, these functions can be called on any architecture. On non-Android architectures, there is no "Android main process". In this case, `android-funcall-in-main-thread-list` first tests whether the variable `*android-main-process-for-testing*` is non-nil (which value must be a process), and if it is sends `func-and-args` to this process by `process-send`. This is based the assumption that this process processes cons events by applying the `cl:car` to the `cl:cdr`, which is the "normal" behavior of the system event processing (that is, what `general-handle-event` does). If you set this variable, make sure that this process processes events in this way. If `*android-main-process-for-testing*` is nil, `android-funcall-in-main-thread-list` arranges for the idle process to apply the `cl:car` to the `cl:cdr`.

Notes

`android-funcall-in-main-thread-list` always queues the function, even if it runs on the main thread. If you need to execute immediately when running on the main thread, check first using `android-main-thread-p`. 
android-get-current-activity  Function

Summary  Return the current activity that was set by the Java method
com.lispworks.Manager.setCurrentActivity.

Package  hcl

Signature  android-get-current-activity => result

Arguments  None.

Values  result  An object of class android.app.Activity, or nil.

Description  The function android-get-current-activity returns the
current activity that was set by the Java method com.lisp-
works.Manager.setCurrentActivity, if the current thread
is the Android main thread.

android-get-current-activity first checks whether the
current thread is the main thread, and if it is not returns nil.
Otherwise, it returns the activity that was last set by
com.lispworks.Manager.setCurrentActivity (an object
of class android.app.Activity). This is the object that is
needed to create dialogs.

Notes  The main purpose of android-get-current-activity is to
decide whether the current code can raise dialogs, and if so to
get the activity to use as a context.

Examples  (example-edit-file "android/dialog")

See also  *android-main-process-for-testing*
android-main-thread-p
Chapter 16, “Android interface”
See also android-main-thread-p
    com.lispworks.Manager.setCurrentActivity

*android-main-process-for-testing*  
Variable

Summary Variable defining the "Android main process" when not running on Android.

Package hcl

Initial value nil

Description The variable *android-main-process-for-testing* defines the "Android main process" when not running on Android. *android-main-process-for-testing* defaults to nil. If it is set, it must be a mp:process object, which processes events which are a cons by applying the cl:car to the cl:cdr.

*android-main-process-for-testing* is used by android-funcall-in-main-thread-list and android-funcall-in-main-thread when they are called on non-Android platforms.

Notes general-handle-event processes conses by applying the cl:car to the cl:cdr, and therefore any process that uses it to process events will do the right thing. That includes the CAPI events loop, and users of wait-processing-events and process-all-events.

android-main-thread-p  
Function

Summary Tests whether the current thread is the Android main (GUI) thread.
**android-main-thread-p**

**Package**: hcl  
**Signature**: `android-main-thread-p => result`  
**Arguments**: None.  
**Values**  
result A boolean.  

**Description**  
The function `android-main-thread-p` is the predicate for whether the current thread is the Android main (GUI) thread. For testing, on non-Android platforms `android-main-thread-p` checks whether the current Lisp process is `*android-main-process-for-testing*` (if this variable is non-nil) or the Idle process (if `*android-main-process-for-testing*` is nil).

**See also**  
`android-funcall-in-main-thread`  
`*android-main-process-for-testing*`

---

**any-capi-window-displayed-p**

**Function**

**Summary**  
A predicate for whether any CAPI window is currently displayed.

**Package**: hcl  
**Signature**: `any-capi-window-displayed-p => result`  
**Values**  
result A boolean.  

**Description**  
The function `any-capi-window-displayed-p` is a predicate for whether any CAPI window (other than dialogs) is currently displayed.
Notes

1. See the CAPI User Guide and Reference Manual for a description of the CAPI toolkit which allows you to write graphical user interfaces in Lisp.

2. Tools in the LispWorks IDE are all CAPI windows.

array-single-thread-p

Function

Summary
The predicate for single-thread arrays.

Package
hcl

Signature
array-single-thread-p array => result

Arguments
array An array.

Values
result A boolean.

Description
The function array-single-thread-p is the predicate for whether an array is one known to be only accessed in a single thread context, as created by

(make-array ... :single-thread t)

or set by set-array-single-thread-p.

See also
make-array
set-array-single-thread-p

array-weak-p

Function

Summary
The predicate for whether an object is a weak array.

Package
hcl

Signature
array-weak-p object => result
Arguments  
object  
A Lisp object.

Values  
result  
A boolean.

Description  
The function \texttt{array-weak-p} returns \texttt{t} if its argument \texttt{object} is a weak array, and otherwise returns \texttt{nil}.

See also  
\texttt{make-array} 
\texttt{set-array-weak}

\textbf{augment-environment}  
\textit{Function}

Summary  
Returns a new environment based on an existing one with different bindings.

Package  
hcl

Signature  
\texttt{augment-environment env &key variable symbol-macro function macro declare reset} ➞ newenv

Arguments  
\texttt{env}  
An environment or \texttt{nil}
\texttt{variable}  
A list of symbols
\texttt{symbol-macro}  
A list of lists
\texttt{function}  
A list of function names
\texttt{macro}  
A list of lists
\texttt{declare}  
A list of declaration-specifiers
\texttt{reset}  
A generalized boolean

Values  
\texttt{newenv}  
An environment

Description  
The function \texttt{augment-environment} returns a new environment \texttt{newenv}, based on \texttt{env} but modified according to the keyword arguments \texttt{variable, symbol-macro, function, macro, declare and reset}. 

855
If \( env \) is nil, then newenv will be based on the null environment. Otherwise, if reset is false (the default) then all of the bindings in \( env \) will be present in newenv unless overridden by the other keyword arguments. Otherwise, if reset is true then all of the non-local bindings in \( env \) will be present in newenv but none of the local bindings will be present. Passing reset as true allows you to create an environment object for calls to variable-information and so on which can access the file compilation environment without seeing local bindings in the lexical environment.

variable should be a list of symbols and newenv will contain these symbols as local variable bindings. A binding will be a special binding if the symbol is declared special non-lexically in \( env \) or a special declaration is present in declare.

symbol-macro should be a list of lists of the form (symbol expansion) and newenv will contain local symbol-macro bindings for each symbol with expansion as its macroexpansion.

function should be a list of function names and newenv will contain these symbols as local function bindings.

macro should be a list of lists of the form (symbol macrofunction) and newenv will contain local macro bindings for each symbol with macrofunction as its macroexpansion function. Each macrofunction is a function of two arguments, a form and an environment, which should return the expanded form.

declare should be a list of declaration-specifiers, which will be added to newenv as if by declare.

It is an error to use a symbol in symbol-macro that is also in variable or is declared special.

It is an error to use a symbol in macro that is also in function.

newenv has the same extent as \( env \), that is it might have dynamic extent within the function that created \( env \).
The lists passed to augment-environment should be not destructively modified afterwards.

Notes
augment-environment is part of the environment access API which is based on that specified in Common Lisp: the Language (2nd Edition).

See also
declaration-information
define-declaration
function-information
map-environment
variable-information

avoid-gc

Function

Summary
Avoids garbage collection if possible in 32-bit LispWorks.

Package
hcl

Signature
avoid-gc => previous-results

Arguments
None.

Values
The function returns the previous settings of minimum-for-sweep, maximum-overflow and minimum-overflow. (see set-gc-parameters for details of these.)

Description
avoid-gc sets various internal parameters so that garbage collection is avoided as far as possible.

This can be useful with non-interactive programs.

If you use avoid-gc, use normal-gc later to reset the parameters to their default settings.

Notes
avoid-gc is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit imple-
mentations. In 64-bit implementations, you can use `set-default-segment-size` to increase the default size of segments in the lower generations (typically generations 0 and 1). This will lead to less frequent garbage collections.

See also  
`gc-if-needed`  
`normal-gc`  
`set-gc-parameters`  
`set-default-segment-size`  
`without-interrupts`  
“Memory Management in 32-bit LispWorks” on page 135

*background-input*  
*background-output*  
*background-query-io*  

Variables

Summary  
Default streams for the standard streams.

Package  
hcl

Initial value  
The value of `cl:*terminal-io*`.

Description  
The variables *background-input*, *background-output* and *background-query-io* are default streams for the standard Common Lisp streams.

These variables are all set to the value of `cl:*terminal-io*` when the image starts, but when the LispWorks IDE starts it sets:

- `*background-output*` to `mp:*background-standard-output*`.
- `*background-input*` to a stream that always returns `EOF`.
- `*background-query-io*` to a stream that interacts with the user using CAPI prompters.
The default value of each of the standard streams is a synonym stream to a *background-* stream:

- *standard-input* is a synonym to *background-input*.
- *standard-output*, *trace-output* and *error-output* are synonyms to *background-output*.
- *query-io* and *debug-io* are synonyms to *background-query-io*.

Thus when the LispWorks IDE is running, output to the standard output streams goes to the mp:*background-standard-output*, and can be viewed in the Output tab of the Listener and Editor windows, and the Output Browser tool. Trying to read from *standard-input* once the environment is running returns EOF. Using *query-io* causes on-screen prompting.

The main purpose of these variables is to redirect the standard input and output streams once the LispWorks IDE is running, because writing to cl:*terminal-io* is not useful in most cases.

You can set or rebind these variables if required, and this changes the default destination of the standard streams.

**Notes**

Processes that are created by CAPI for an interface while the IDE is running rebind the standard input, output and query I/O streams to themselves (so setting them in these processes does not change the global value). This does not happen on processes that are not created by CAPI, and does not happen when the LispWorks IDE is not running, in particular in delivered applications. When the LispWorks IDE is running, the output to standard output stream on other processes will still go by default to the mp:*background-standard-output*, because *background-output* is set to it.
These variables were new in LispWorks 7.0.

In LispWorks 6.1 and earlier versions, CAPI processes in the LispWorks IDE bound the output streams to `mp:*background-standard-output*`, the standard input to a stream that returns EOF and *QUERY-IO* to a stream that interacts with the user using CAPI prompters. Hence, for these processes, the default behavior has not changed. However input and output on other processes was going to/from the `cl:*terminal-io*` by default, which caused various problems. The main purpose of these variables is to fix these problems.

See also `*standard-input*`

### binds-who

**Function**

**Summary** Lists special variables bound by a definition.

**Package** hcl

**Signature** `binds-who function => result`

**Arguments**

- `function` A symbol or a function dspec.

**Values**

- `result` A list.

**Description** The function `binds-who` returns a list of the special variables bound by the definition named by `function`.

**Notes** The cross-referencing information used by `binds-who` is generated when code is compiled with source-level debugging switched on.

**See also** `references-who`

`sets-who`
toggle-source-debugging
who-binds

block-promotion

Macro

Summary
Prevents promotion of objects into generation 2 during the execution of body.

Package
hcl

Signature
block-promotion &body body => result

Arguments
body
Forms executed as an implicit prog

Values
result
The result of evaluating the final form in body.

Description
The macro block-promotion executes body and prevents promotion of objects into generation 2 during this execution. After body is executed, generations 0 and 1 are collected.

This is useful when a significant number of transient objects actually survive all the garbage collections on generation 1. These would normally then be promoted and, by default, never get collected. In such a situation, (gc-generation t) will free a large amount of space in generation 2.

block-promotion can be thought of as doing set-promotion-count on generation 1 with an infinite count, for the duration of body.

block-promotion is suitable only for use in particular operations that are known to create such relatively long-lived, but transient, objects. In typical uses these are objects that live for a few seconds to several hours. An example usage is Lisp-Works compile-file, to ensure the transient compile-time data gets collected.
block-promotion has global scope and hence may not be useful in an application such as a multithreaded server. During the execution of body, generation 1 grows to accommodate all the allocated data, which may have some negative effects on the behavior of the system, in particular on its interactive response.

Notes
1. Symbols and process stacks are allocated in generation 2 or 3 (see *symbol.alloc-gen-num*) hence block-promotion cannot prevent these getting into that generation. allocation-in-gen-num can also cause allocation in higher generations.
2. In 64-bit LispWorks, block-promotion is implemented using set-blocking-gen-num.

See also allocation-in-gen-num mark-and-sweep set-promotion-count

**building-universal-intermediate-p**

Function

Summary
Deprecated. Simply returns nil.

Package
hcl

Signature
building-universal-intermediate-p => intermediatep

Arguments
None

Values
intermediatep A boolean.

Description
The function building-universal-intermediate-p is deprecated. The return value intermediatep is nil.

Compatibility note
In LispWorks 6.1 for Macintosh and earlier versions, building-universal-intermediate-p is used in a build script to
determine whether LispWorks is building an intermediate image when making a universal binary.

In LispWorks 7.0 and later versions universal binaries are not supported, and this function always returns `nil`, on all platforms.

See also `save-universal-from-script`
`save-argument-real-p`

### calls-who

**Function**

**Summary**
Lists functions called by a function.

**Package**
hcl

**Signature**
calls-who dspec => callees

**Arguments**
dspec A dspec.

callees A list.

**Description**
The function `calls-who` returns a list of the dspecs naming the functions called by the function named by `dspec`.

See also the editor commands List Callees, and Show Paths From.

**Notes**
The cross-referencing information used by `calls-who` is generated when code is compiled with source-level debugging switched on.

**Example**

```
(calls-who '(method foo (string)))
```

See also `toggle-source-debugging`
`who-calls`
### cd

**Macro**

**Summary**
Changes the current directory.

**Package**
hcl

**Signature**
`cd &optional directory => current-dir`

**Arguments**
directory
A pathname designator specifying the new directory.

**Values**
current-dir
A physical pathname.

**Description**
The macro `cd` changes the current directory to that specified by `directory`. `directory` may be an absolute or relative pathname, and defaults to the string `"~/*`.

**Notes**
`cd` should not be used in multithreaded applications. In general we discourage you from using it.

**See also**
change-directory
get-working-directory

---

### change-directory

**Function**

**Summary**
Changes the current directory.

**Package**
hcl

**Signature**
`change-directory directory => current-dir`

**Arguments**
directory
A pathname designator specifying the new directory.

**Values**
current-dir
A physical pathname.
Description: The `change-directory` function changes the current directory to that specified by `directory`. `directory` may be an absolute or relative pathname.

Use the `get-working-directory` function to find the current directory.

Notes: The `change-directory` function should not be used in multithreaded applications. In general, we discourage you from using it.

See also: `cd`, `get-working-directory`

---

**check-fragmentation**

Function

Summary: Provides information about the fragmentation in a generation in 32-bit LispWorks.

Package: `hcl`

Signature: `check-fragmentation gen-num => total-free, total-small-blocks, total-large-blocks`

Arguments: `gen-num` - 0 for the most recent generation, 1 for the most recent two generations, and so on up to a maximum (usually 3). Numbers outside this range signal an error.

Values: `total-free` - Total free space in the generation.

`total-small-blocks` - Amount of free space in the generation which is available in blocks of 512 bytes or larger.

`total-large-blocks` - Amount of free space in the generation which is available in blocks of 4096 bytes or larger.
Description  The latter two values give indication of the level of fragmentation in the generation. This information can be used, for example, to decide whether to call `try-move-in-generation`.

Notes  `check-fragmentation` is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations, where `gen-num-segments-fragmentation-state` is available instead.

See also  `try-compact-in-generation`  `try-move-in-generation`  “Guidance for control of the memory management system” on page 130

**clean-down**  
*Function*

**Summary**  Frees memory and reduces the size of the image, if possible.

**Package**  hcl

**Signature**  `clean-down &optional full => new-size`

**Arguments**  `full`  A generalized boolean.

**Values**  `new-size`  A positive integer.

**Description**  Tries to free as much memory as possible and then reduce the size of the image as much as possible, and also move all the allocated objects to an old generation.

`full` controls whether to operate on the highest generation. The default value of `full` is `t`.

If `full` is `t`, `clean-down` does a mark and sweep on generation 3, promotes all the objects into generation 3, deletes the...
empty segments and tries to reduce the image size. This is called by default before saving an image.

If `full` is `nil`, `clean-down` does a mark and sweep on generation 2, promotes all the objects to generation 2 and tries to reduce the size of all generations up to 2, but does not touch generation 3.

`clean-down` returns the new size of the Lisp image after reduction, in bytes.

`clean-down` may fail to delete empty segments if there are static segments in high address space.

### Notes

1. `try-move-in-generation` (which is implemented only in 32-bit LispWorks) uses less CPU than `clean-down`, though it does not do the mark and sweep.

2. In 64-bit LispWorks, `clean-down` is implemented as if by `(gc-generation 7 :coalesce t)` though you can use `gc-generation` directly for better control.

3. In the Mobile GC, `clean-down` performs the same GC as `(gc-generation t)`.

4. `clean-down` may temporarily increase memory usage, and when called with `full nil` may result in a larger Lisp image (though better organized, and therefore behaving better). In 32-bit LispWorks in situations where it is important not to increase memory usage, such as when the operating system signals that memory is low, use `reduce-memory` instead.

### See also

- `gc-generation`
- `reduce-memory`
- `save-image`
- `try-move-in-generation`

“Guidance for control of the memory management system” on page 130
clean-generation-0

Function

Summary  Attempts to promote all objects from generation zero into generation one, thereby clearing generation zero, in 32-bit LispWorks.

Package  hcl

Signature  clean-generation-0 => 1

Arguments  None

Values  Returns the value 1.

Description  This is useful when passing from a phase of creating long-lived data to a phase of mostly ephemeral data, for example, the end of loading an application and the start of its use.

Notes  1. The function may not be very useful, as it may be more efficient to directly allocate the objects in a particular generation in the first place, using allocation-in-gen-num or set-default-generation.

2. clean-generation-0 is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations, where the same effect can be obtained by a call (gc-generation 0).

Example  ; allocate lots of non-ephemeral objects
; .......
(clean-generation-0)

See also  allocation-in-gen-num
collect-generation-2
collect-highest-generation
expand-generation-1
gc-generation
clear-code-coverage
reset-code-coverage
restore-code-coverage-data

Summary
Modify the internal code-coverage-data object.

Package
hcl

Signature

clear-code-coverage => result
reset-code-coverage => result

restore-code-coverage-data code-coverage-data &key error => result

Arguments
code-coverage-data
A code-coverage-data object.

error :warn, nil or t.

Value
result A boolean.

Description
The function clear-code-coverage clears the internal code-coverage-data object, which means removing all the files from it, so that their counters are not accessible anymore. Note that it does not actually remove the counters from the code.

The function reset-code-coverage resets all the counters in the internal code-coverage-data object to 0.

The function restore-code-coverage-data sets the counters of all files that appear in both the internal code coverage data and the argument code-coverage-data to the counters in the code-coverage-data argument. All these files
need to have the same code coverage code, that is they must be based on the same binary file.

`error` controls what happens for files that do not have the same code coverage code. Value `:warn` means warn and continue, `nil` means quietly skip it, and `t` means signal an error. `restore-code-coverage-data` never restores a file with no matching code coverage code. The default value of `error` is `:warn`.

The value of `result` indicates whether there was an internal `code-coverage-data` object when the function was called.

### Notes

1. If `error` is `t`, some of the files would be restored and some not, leaving the internal `code-coverage-data` object in an inconsistent state.

2. All these functions also reset any snapshot by calling `reset-code-coverage-snapshot`.

### See also

Chapter 10, “Code Coverage”

### code-coverage-data

**Type**

**Summary**

A structure containing information about code coverage.

**Package**

hcl

**Readers**

code-coverage-data-name

code-coverage-data-create-time

**Description**

The type `code-coverage-data` is a structure containing information about code coverage.

`code-coverage-data` contains information about some set of files. With the exception of the internal code coverage data, `code-coverage-data` does not change after it is created. The internal code coverage data contains information about all
files that have been loaded with code coverage (since the last call to `clear-code-coverage`). A file is "with code coverage" when it is a binary file compiled with code coverage on (see `generate-code-coverage`).

The counters in the internal code coverage data are the counters that the actual code is referencing, and therefore they are modified whenever any of this code is executing. For each file the counters are either actual counters or binary flags (see `counters` argument in `generate-code-coverage`), but inside the structure there may be files of either counter type.

All other `code-coverage-data` structures start their life as copies of the internal code coverage data, and then they can be further manipulated. They are displayed by `code-coverage-data-generate-coloring-html` or the LispWorks IDE.

The `name` is supplied to the data when it is created by functions like `copy-current-code-coverage`, and the `create-time` is the universal time when the data was created. These values are provided so that you can track your data: they are not used by the system. `name` can be any Lisp object, but normally should be a symbol or a string (because if you save the data name will be written too, so it is best if does not point to a large structure).

See also Chapter 10, “Code Coverage”

```
\[\text{code-coverage-data-generate-coloring-html} \quad \text{Function}\]
```

Summary

Generates HTML showing the code coverage.

Package `hci`
Signature  
\texttt{code-coverage-data-generate-coloring-html target &key}
\texttt{code-coverage-data shared-source-directory filter target-type color-
uncovered color-covered show-counters counter-space index-filename
index-name index-sort index-mark-not-entered index-mark-partial
index-show-non-runtime open}

Arguments  
\texttt{target} A pathname designator.
\texttt{code-coverage-data} 
A \texttt{code-coverage-data} object.
\texttt{shared-source-directory} 
A pathname designator.
\texttt{filter} A string, a function or a symbol naming a function.
\texttt{target-type} A string or \texttt{nil}.
\texttt{color-uncovered} A boolean.
\texttt{color-covered} A boolean.
\texttt{show-counters} A boolean.
\texttt{counter-space} \texttt{nil, :before, :after, :both or t}.
\texttt{index-filename} A pathname designator or \texttt{nil}.
\texttt{index-name} A string.
\texttt{index-sort} One of the keywords \texttt{:relative-name, :name} and \texttt{:uncovered}.
\texttt{index-mark-not-entered} 
A boolean.
\texttt{index-mark-partial} 
A boolean.
\texttt{index-show-non-runtime} 
A boolean.
\texttt{open} A boolean.
The function `code-coverage-data-generate-coloring-html` generates HTML showing the code coverage.

`target` specifies the directory for the HTML files, and optionally the name of the index file, if `target` has a name component and `index-filename` is not supplied.

`code-coverage-data` must be a `code-coverage-data` object to use. Otherwise `code-coverage-data-generate-coloring-html` uses the internal data.

`shared-source-directory` must specify a directory path. It has two effects:

- HTML is produced only for source files in the `shared-source-directory` (the `filter` may exclude some of these), and

- The path of each HTML file is constructed from the relative path of the source file with respect to the `shared-source-directory` (as produced by `cl:enough-namestring`) merged with the `target` directory. The result is a tree of HTML files which is parallel to the tree of the source files.

If `shared-source-directory` is not supplied, all files that passed the `filter` are produced, and the target HTML file has the filename of the source file inside the `target` directory. Note that this may cause clashes if there are files with the same name in the data.

`filter` can be used to restrict which files HTML is produced for. If `filter` is a string it is interpreted as a regexp. If the `cl:namestring` of the truename of a source file matches `filter` (as by `find-regexp-in-string`) then HTML is produced for this source file. If `filter` is a function (or fbound symbol) it must take two arguments, the truename and the `code-coverage-file-stats` for this source file, and return a boolean specifying whether to produce HTML for this source file. The stats object can be accessed by the `code-coverage-file-stats` accessor functions (for example `code-coverage-file-stats-lambdas-count`). If `filter` is not supplied, all
files (or, if `shared-source-directory` is supplied, all those files inside it) are produced.

`target-type` specifies the type of the output files. The default value of `target-type` is "htm".

`color-uncovered`, `color-covered`, `show-counters` and `counter-space` control the HTML output. See “Source files HTML coloring” below for details. Note that the colors to actually use are specified by `code-coverage-set-html-background-colors`.

`color-uncovered` controls whether uncovered forms are colored. These include forms that did not execute at all, eliminated forms and forms which were partially executed but the unexecuted part is hidden (in a macroexpansion). The default value of `color-uncovered` is `t`.

`color-covered` controls whether covered forms are colored. These include forms that were fully executed, and those parts of partially executed forms that were executed. The default value of `color-covered` is `nil`.

`show-counters` controls whether to insert counters in the HTML. The default value of `show-counters` is `t`.

`counter-space` specifies whether to insert a space before and/or after each counter. The value `t` has the same meaning as `:both`. The default value of `counter-space` is `:after`.

`index-filename`, `index-name`, `index-sort`, `index-mark-not-entered`, `index-mark-partial` and `index-show-non-runtime` control the generation of the index file. See “Index file” below for the description of the index file’s contents.

`index-filename`, when supplied, specifies the name of the index file. It is merged with the `target` path to generate the full path. Note that the file type should be included in either `index-filename` or the `target` path. If `index-filename` is not supplied, it defaults to "code-coverage-index.htm". If `index-filename` is `nil`, no index file is produced.
index-name is printed (with format directive ~A) as part of the title of the index file, and not used otherwise. The default value of index-name is "Index".

index-sort controls the order files are listed in the table in the index. :relative-name means sort by the relative name of the source file with respect to shared-source-directory. If shared-source-directory is not supplied, :relative-name has the same effect as :name. :name means sort by the name of the source file. :uncovered means sort by the number of not fully covered run time lambdas in the file (the sum of code-coverage-file-stats-not-called and code-coverage-file-stats-partially-covered called with :runtime). The default value of index-sort is :relative-name.

index-mark-not-entered controls whether to mark cells in the run time part for uncovered lambdas. The default value of index-mark-not-entered is t.

index-mark-partial controls whether to mark cells in the run time part for lambdas that are partially covered. The default value of index-mark-partial is t.

index-show-non-runtime controls whether to show the non-run time part of the table. The default value of index-show-non-runtime is t.

open specifies whether the index file should opened (by open-url) once it is generated. The default value of open is nil.

The HTML output

code-coverage-data-generate-coloring-html generates a HTML file for each source file in code-coverage-data that is inside the shared-source-directory (or all source files if shared-source-directory is nil) and pass the filter (or all if filter is nil), as described above, and one index file with statistics. It uses background colors to mark various things (see below), and these colors can be set by code-coverage-set-html-back-
The colors that are described below are the default colors.

Index file

The index file contains a table with a single row per file.

The first column shows the file "relative name", which is relative to the optional `shared-source-directory`, or just the filename. The rest of the columns contain statistics, which are divided into 2 parts: run time lambdas and optional non-run time lambdas. "Lambda" here means a separate piece of code (for example code that is called inline does not count as a separate lambda). Run time lambdas refer to code that is expected to run at run time, which includes things like functions and methods. Non-run time lambdas are other lambdas, like macros and top-level forms (as known as one-shot forms). More accurately, run time and non-run time refer to the counts which are returned by the `code-coverage-file-stats` accessor functions (for example `code-coverage-file-stats-lambdas-count`) when they are called with `:runtime` or `:non-runtime`. See `code-coverage-file-stats` for details.

The run time and the non-run time parts each contain 4 columns:

- **Total**: The total number of lambdas, as returned by `code-coverage-file-stats-lambdas-count`.
- **Full**: The number of lambdas that were fully covered, as returned by `code-coverage-file-stats-fully-covered`.
- **Partial**: The number of lambdas that were partially covered, as returned by `code-coverage-file-stats-partially-covered`.

- **Index file contains a table with a single row per file.**
- **The first column shows the file "relative name", which is relative to the optional `shared-source-directory`, or just the filename.**
- **The rest of the columns contain statistics, which are divided into 2 parts: run time lambdas and optional non-run time lambdas. "Lambda" here means a separate piece of code (for example code that is called inline does not count as a separate lambda).**
- **Run time lambdas refer to code that is expected to run at run time, which includes things like functions and methods. Non-run time lambdas are other lambdas, like macros and top-level forms (as known as one-shot forms). More accurately, run time and non-run time refer to the counts which are returned by the `code-coverage-file-stats` accessor functions (for example `code-coverage-file-stats-lambdas-count`) when they are called with `:runtime` or `:non-runtime`. See `code-coverage-file-stats` for details.**
- **The run time and the non-run time parts each contain 4 columns:**
  - **Total**: The total number of lambdas, as returned by `code-coverage-file-stats-lambdas-count`.
  - **Full**: The number of lambdas that were fully covered, as returned by `code-coverage-file-stats-fully-covered`.
  - **Partial**: The number of lambdas that were partially covered, as returned by `code-coverage-file-stats-partially-covered`.
None The number of lambdas that were not covered, as returned by `code-coverage-file-stats-not-called`.

In the run time columns, Partial and None cells which are non-zero are optionally marked with a colored background. This helps you to see which files contain run time forms that were not executed. The default color is DarkSalmon, and this can be set by `code-coverage-set-html-background-colors` with keyword argument `marked-cell`.

**Source files HTML coloring**

The HTML file corresponding to a source file contains the full text of the source file (including comments), with parts optionally highlighted by background colors, and optional counters and some text added. At the time of writing, the default behavior is to highlight uncovered forms and add counters. The background colors can be changed by `code-coverage-set-html-background-colors`. The general issues associated with coloring are covered in “Understanding the code coverage output” on page 124.

**Notes**

If no file containing code coverage code was loaded, there is no internal data, so if `code-coverage-data` is not supplied then `code-coverage-data-generate-coloring-html` signals an error.

**See also**

- `code-coverage-data`
- `code-coverage-set-html-background-colors`
- Chapter 10, “Code Coverage”
- “Understanding the code coverage output” on page 124

### code-coverage-data-generate-statistics

**Function**

**Summary**

Generates statistics about code coverage.
**Package**

hcl

**Signature**

code-coverage-data-generate-statistics &key code-coverage-data sort => result

**Arguments**

code-coverage-data

A code-coverage-data object.

sort

A generalized boolean.

**Values**

result

A vector of code-coverage-file-stats objects.

**Description**

The function code-coverage-data-generate-statistics generates statistics about code coverage. code-coverage-data, if supplied, must be a code-coverage-data object, otherwise the internal code-coverage-data object is used. For each file in the data, code-coverage-data-generate-statistics generates a code-coverage-file-stats object. It returns a vector of these code-coverage-file-stats objects.

If the argument sort is non-nil (the default), the vector is sorted by the cl:file-namestring of the source file.

**Notes**

1. The stats objects do not change after code-coverage-data-generate-statistics returns, even if the data that was used is the internal one.

2. The statistics are only coverage, that is they treat the counters as binary zero/non-zero values. That includes negative counters (which may occur if the supplied data is a result of subtraction), which counted as "Covered".

3. The stats objects can be accessed by the code-coverage-file-stats readers.

**See also**

code-coverage-data

code-coverage-file-stats
**code-coverage-file-stats**

**Type**

**Summary**
A structure containing code coverage statistics.

**Package**
hcl

**Readers**
code-coverage-file-stats-source-file

code-coverage-file-stats-lambdas-count
code-coverage-file-stats-called
code-coverage-file-stats-fully-covered
code-coverage-file-stats-hidden-covered
code-coverage-file-stats-not-called
code-coverage-file-stats-partially-covered
code-coverage-file-stats-counters-count
code-coverage-file-stats-counters-executed
code-coverage-file-stats-counters-hidden

**Description**
Objects of type **code-coverage-file-stats** are created by **code-coverage-data-generate-statistics**, and are then accessed by the readers.

code-coverage-file-stats-source-file returns the true-name of the source file. The pseudo-readers return integers counting "lambdas" in this file, where "lambda" here means a separate function object that was produced by the compiler. In most cases these correspond to pieces of code that you can see, like a function that results from **cl:defun** or **cl:defmethod**, or a lambda that appears in your code, but in some cases the compiler generates functions in a non-obvious way.

The pseudo-readers each have signature

pseudo-reader ccfs keyword => count

where ccfs is a **code-coverage-file-stats** object, and keyword specifies the kind of lambda. All lambdas belong to one of these four kinds:

:functions Functions that are defined by **cl:defun**.
Macros and macro-like (for example cl:defsetf).

Load time lambdas that the compiler generates.

Other lambdas (including cl:defmethod).

In addition, the following three values of keyword can be used:

All lambdas.

:runtime :functions and :lambdas.

:non-runtime :one-shot and :macros.

Each pseudo-reader returns the number of lambdas or counters in the file of the kind specified by the keyword. These are:

code-coverage-file-stats-lambdas-count
All lambdas.

code-coverage-file-stats-called
Lambdas that have been called.

code-coverage-file-stats-fully-covered
Lambdas which were fully covered, that is all of their counters are non-zero.

code-coverage-file-stats-hidden-covered
Lambdas where there are counters which are 0, but do not correspond to actual source code (result of macroexpansion).

code-coverage-file-stats-not-called
Lambdas that were not called at all.

code-coverage-file-stats-partially-covered
Lambdas that were partially covered, but part of the source did not execute.

code-coverage-file-stats-counters-count
All counters.

**code-coverage-file-stats-counters-executed**

Counters that executed (that is, they are not zero).

**code-coverage-file-stats-counters-hidden**

Counters which have not been executed and are hidden, that is not in the source (in a result of macroexpansion).

**Notes**

1. The statistics are based on interpreting the counters as a binary switch of zero/non-zero. Negative counter values (which may occur if the code coverage data is a result of a subtraction operation such as `subtract-code-coverage-data`) are interpreted as "executed".

2. The run time/non-run time distinction is intended to correspond to code that would run in the actual application (run time) and code that is used only at compile-time or load-time.

**Examples**

```lisp
(code-coverage-file-stats-called code-coverage-file-stats :runtime => lambda-count
```

where `lambda-count` is the number of lambdas in the file which are "run time" and have been called.

**See also**

Chapter 10, “Code Coverage”

**code-coverage-set-editor-colors**

*Function*

**Summary**

Specifies the colors that the editor uses to color code coverage.

**Package**

`hcl`
The HCL Package

Signature

```
(code-coverage-set-editor-colors &key counters counters-negative uncovered partially-covered fully-covered hidden-partial error warn eliminated)
```

Arguments

- **counters**: A CAPI color or an editor face.
- **counters-negative**: A CAPI color or an editor face.
- **uncovered**: A CAPI color or an editor face.
- **partially-covered**: A CAPI color or an editor face.
- **fully-covered**: A CAPI color or an editor face.
- **hidden-partial**: A CAPI color or an editor face.
- **error**: A CAPI color or an editor face.
- **warn**: A CAPI color or an editor face.
- **eliminated**: A CAPI color or an editor face.

Description

The function `code-coverage-set-editor-colors` changes the colors or faces that the editor uses to color code coverage.

Each argument is a CAPI color name, color alias or color specification, or an `editor:face` object (the result of `editor:make-face`). See "The Color System" in the CAPI User Guide and Reference Manual for details about CAPI colors.

When an argument value is an `editor:face`, it specifies the face to use. Otherwise, it specifies the background color to use.

The faces and colors are used to color parts of the code as in `code-coverage-set-html-background-colors`. Note that `code-coverage-set-editor-colors` does not accept a :marked keyword argument like `code-coverage-set-html-background-colors` does.

See also

- `code-coverage-set-html-background-colors`
**Function**

**code-coverage-set-editor-default-data**

**Summary**
Sets the default code coverage data that the editor uses when coloring.

**Package**
hcl

**Signature**
code-coverage-set-editor-default-data object

**Arguments**
object     A code-coverage-data object, a string, a pathname or nil.

**Description**
The function `code-coverage-set-editor-default-data` sets the default code coverage data that the editor uses when it colors a file.

If `object` is a code-coverage-data object, this is used as-is. If `object` is a string or pathname then it should name a file that was created by `save-current-code-coverage` or `save-code-coverage-data`. The data is loaded from this file using `load-code-coverage-data` and used.

If `object` is nil then `code-coverage-set-editor-default-data` uses the internal code coverage data. The default value of `object` is nil.

**Notes**
The editor commands Code Coverage File and Code Coverage Current Buffer use this data.

**See also**
code-coverage-data
save-current-code-coverage
save-code-coverage-data
load-code-coverage-data
Code Coverage File in the Editor User Guide
Code Coverage Current Buffer in the Editor User Guide
Code Coverage Set Default Data in the Editor User Guide
Code Coverage Load Default Data in the Editor User Guide

**code-coverage-set-html-background-colors**  
*Function*

**Summary**
Sets the background colors used in the HTML code coverage output.

**Package**
hcl

**Signature**
code-coverage-set-html-background-colors &key counters
counters-negative uncovered partially-covered fully-covered hidden-partial error warn eliminated marked-cell

**Arguments**
counters A string.
counters-negative A string.
uncovered A string.
partially-covered A string.
fully-covered A string.
hidden-partial A string.
error A string.
warn A string.
eliminated A string.
marked-cell A string.

**Description**
The function code-coverage-set-html-background-colors sets the background colors that code-coverage-data-generate-coloring-html uses to color the output.

Each of the keyword arguments, when supplied, must specify a color that is valid HTML. This can be either:
• A hexadecimal value "#RRGGBB" where RR, GG and BB are hexadecimal numbers specifying the Red, Green and Blue values, or
• A name that web browsers recognize.

LispWorks does not actually check that the name is a known name.

Only those colors for which a keyword argument is supplied are affected.

See “Understanding the code coverage output” on page 124 for details of how the colors are used.

See also code-coverage-data-generate-coloring-html
code-coverage-set-editor-colors
“Understanding the code coverage output” on page 124

**collect-generation-2**

*Function*

**Summary**
Controls whether generation 2 is garbage collected in 32-bit LispWorks.

**Package**
hcl

**Signature**
collect-generation-2 on => size

**Arguments**
on
If on is nil, generation 2 is not garbage collected. If on is t, the generation is garbage collected.

**Values**
size
The current size of the image.

**Description**
Controls whether generation 2 is garbage collected. (Generation 2 normally holds long-lived objects created dynamically.)
**collect-highest-generation**

*Function*

**Summary**
Controls whether the top generation is garbage-collected in 32-bit LispWorks.

**Package**
hcl

**Signature**
collect-highest-generation flag

**Arguments**
flag
If flag is non-nil, the top generation is collected; if flag is any other value, the top generation is not collected. The default is nil.

**Values**
collect-highest-generation returns no values.

**Description**
The function collect-highest-generation controls whether the top generation is garbage-collected in 32-bit LispWorks.

**Notes**
collect-highest-generation is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations.
See also

avoid-gc
clean-generation-0
collect-generation-2
expand-generation-1
normal-gc

“Memory Management in 32-bit LispWorks” on page 135

*compiler-break-on-error*

Variable

Summary
Controls whether compile-file handles compilation errors.

Package
hcl

Initial value
nil

Description
If an error occurs during compilation of a form by compile-file, an error handler normally causes the compilation of that form to be skipped, and the error is reported later.

When *compiler-break-on-error* is non-nil, an error during compilation by compile-file is signaled and the debugger is entered.

See also
compile-file

compile-file-if-needed

Function

Summary
Compiles a Lisp source file if it is newer than the corresponding fasl file.

Package
hcl

Signature
compile-file-if-needed input-pathname &key output-file load &allow-other-keys => output-truename, warnings-p, failure-p

Arguments
input-pathname A pathname designator.
The function `compile-file-if-needed` compares the `file-write-date` of the source file named by `input-pathname` with the `file-write-date` of the appropriate fasl file (as computed by `compile-file-pathname` from `input-pathname` and `output-file`).

If the fasl file does not exist or is older than `input-pathname`, then `compile-file` is called with `input-pathname, output-file, load` and any other arguments passed., and the values returned are those returned from `compile-file`.

Otherwise, if `load` is true `compile-file-if-needed` loads the fasl file and returns `nil`, and if `load` is `nil` it simply returns `nil`.

**output-file**  A pathname designator.

**load**      A generalized boolean.

**Values**

**output-truename**  A pathname or `nil`.

**warnings-p**    A generalized boolean.

**failure-p**    A generalized boolean.

**Description**

The function `compile-file-if-needed` compares the `file-write-date` of the source file named by `input-pathname` with the `file-write-date` of the appropriate fasl file (as computed by `compile-file-pathname` from `input-pathname` and `output-file`).

If the fasl file does not exist or is older than `input-pathname`, then `compile-file` is called with `input-pathname, output-file, load` and any other arguments passed., and the values returned are those returned from `compile-file`.

Otherwise, if `load` is true `compile-file-if-needed` loads the fasl file and returns `nil`, and if `load` is `nil` it simply returns `nil`. 
Example

CL-USER 19 > (compile-file-if-needed "H:/tmp/foo.lisp"
   :output-file
   "C:/temp/"

;;; Compiling file H:/tmp/foo.lisp ...
;;; Safety = 3, Speed = 1, Space = 1, Float = 1,
Interruptible = 0
;;; Compilation speed = 1, Debug = 2, Fixnum safety = 3
;;; Source level debugging is off
;;; Source file recording is on
;;; Cross referencing is off
;; (TOP-LEVEL-FORM 1)
;; (TOP-LEVEL-FORM 2)
;; (TOP-LEVEL-FORM 3)
;; FOO
;; BAR
#P"C:/temp/foo.ofasl"
NIL
NIL

CL-USER 20 > (compile-file-if-needed "H:/tmp/foo.lisp"
   :output-file
   "C:/temp/"
   :load t)

;;; Loading fasl file C:\temp\foo.ofasl
NIL
NIL

See also

compile-file

Functions

copy-code-coverage-data

Copy, save and load code-coverage-data objects.

Package

hcl

Signature

copy-code-coverage-data ccd name => new-ccd
copy-current-code-coverage &optional name => new-ccd
load-code-coverage-data pathname &key errorp => ccd
save-code-coverage-data pathname ccd => t
save-current-code-coverage pathname &optional name => t

Arguments
ccd A code-coverage-data object.
name A Lisp object, normally a symbol or a string.
pathname A pathname designator.

Values
new-ccd A code-coverage-data object.

Description
The function copy-code-coverage-data copies its ccd argument. The copy is deep, such that ccd and new-cdd do not share data, except read-only objects like pathnames. name is the name supplied to the new copy.

The function copy-current-code-coverage copies the internal code coverage data. The default value of name is "Copy".

The function save-code-coverage-data saves the code coverage data in pathname. The saving is done in the same binary form that the compiler and dump-forms-to-file use. The data can be loaded by load-code-coverage-data. save-code-coverage-data always saves to a file with type "ccd". If pathname does not have a type, save-code-coverage-data adds the type "ccd". If pathname has another type, save-code-coverage-data signals an error.

The function save-current-code-coverage saves the internal code coverage data. name is the name supplied to the saved data. The default value of name is (pathname-name pathname). Like save-code-coverage-data, save-current-code-coverage always saves to a file with type "ccd".

The function load-code-coverage-data loads code coverage data from pathname and returns it. pathname must name a file that was created by save-code-coverage-data or save-current-code-coverage (with or without the "ccd" type).
errorp determines what to do when load-code-coverage-data fails to load. Value nil means return nil, otherwise it calls error. If errorp is true but not t, when load-code-coverage-data calls error it passes errorp as if it is the name of the function that fails. This can be used to give a better indication which function failed. The default value of errorp is t.

Notes A code-coverage-data object can be also written "by hand" into fasl files using dump-form or dump-forms-to-file. In this case you will need to arrange to recover it when the fasl is loaded. load-code-coverage-data uses load-data-file with a callback.

See also code-coverage-data Chapter 10, "Code Coverage"

copy-to-weak-simple-vector Function

Summary Creates a weak vector with the same contents as the supplied vector.

Package hcl

Signature copy-to-weak-simple-vector vector-t => weak-vector

Arguments vector-t An array of type (vector t).

Values weak-vector A weak array of type (vector t).

Description The function copy-to-weak-simple-vector creates and returns a weak vector with the same contents as the argument vector-t.

Apart from the checking of arguments, this is equivalent to:

(replace (make-array (length vector-t) :weak t) vector-t)
See `set-array-weak` for a description of weak vectors.

See also

- `make-array`
- `set-array-weak`
- “Freeing of objects by the GC” on page 165

### create-macos-application-bundle

#### Function

**Summary**

Creates a Mac OS X application bundle for the running LispWorks image.

**Package**

`hcl`

**Signature**


**Arguments**

- `target-path` A pathname designator.
- `template-bundle` A pathname designator.
- `bundle-name` A string.
- `signature` A string.
- `package-type` A string.
- `extension` A string.
- `application-icns` A pathname designator.
- `identifier` A string.
- `version` A string.
- `build` A string.
- `version-string` A string.
- `help-book-name` A string.
document-types  A list or t.
executable-name  t or nil.

Values  path  A pathname.

Description  The function `create-macos-application-bundle` creates a Mac OS X application bundle for the running LispWorks image, and returns the pathname `path` in which an image is expected to be saved. If you are saving an image, it is convenient to use `save-image-with-bundle`.

target-path is where the new bundle is created.

By default `create-macos-application-bundle` uses the application bundle of the current image as a template, and modifies it according to its arguments. If you do not supply any of the keyword arguments, the only modification is to the actual path.

template-bundle can be supplied to provide a path for an application bundle which will be used as a template. If `template-bundle` is not supplied, `create-macos-application-bundle` uses the path of the bundle of the current image. Except when specified, all the other parameters default to their values in the `template-bundle`.

bundle-name provides CFBundleName. The default value is the name of the last directory component in target-path.

signature is the signature in the PkgInfo file.

package-type is the package type, CFBundlePackageType. The default value of package-type is "APPL".

extension is the extension to add to the last component of target-path. The default value of extension is "app", as in "LispWorks.app".

application-icns provides CFBundleIconFile.

package-type is the package type, CFBundlePackageType. The default value of package-type is "APPL".
extension is the extension to add to the last component of target-path. The default value of extension is "app", as in "Lisp-Works.app".

application-icns provides CFBundleIconFile.

identifier provides CFBundleIdentifier. You must change this if you are creating a bundle for your own application.

version is the version value, CFBundleVersion. If template-bundle is nil, version defaults to the value returned by cl:lisp-implementation-version.

version-string provides CFBundleShortVersionString. If version-string is nil (the default), then version and build (if non nil) are used to make a default string.

help-book-folder provides CFBundleHelpBookFolder.

help-book-name provides CFBundleHelpBookName.

document-types provides the CFBundleDocumentTypes dict array. Each item of the list document-types should be a list of the form (name extensions icns-file os-types role) which provide the dict values as follows: the string name provides CFBundleTypeName; the list of strings extensions provides the contents of the array CFBundleTypeExtensions; the pathname designator icns-file provides the string CFBundleTypeIconFile; the list of strings os-types provides the contents of the array CFBundleTypeOSTypes and the string role provides CFBundleTypeRole. role can be omitted and defaults to "Editor". os-types can be omitted and defaults to ("****"). The default value of document-types is t, which means copy them from the application bundle template-bundle.

executable-name is the filename of the LispWorks image executable, not including the directory. The default value of executable-name is the pathname name of the last component of target-path.

Notes

create-macos-application-bundle is implemented only in LispWorks for Macintosh.
create-temp-file
open-temp-file

Summary
Creates a "temp file" and returns a pathname or a stream to it.

Package
hcl

Signature
create-temp-file &key file-type directory prefix => pathname

Signature
open-temp-file &key file-type element-type directory prefix
delete-when-close external-format => stream

Arguments
file-type A string or nil.

element-type A type specifier.

directory A pathname designator.

prefix A string or nil.

delete-when-close A generalized boolean.

external-format An external file format designator.

Values
pathname A pathname.

stream An I/O stream.

Description
The function open-temp-file opens a "temp file". This is a new file in the "temp directory" which is guaranteed to be new. Its name contains a random element. The permissions of the file are read-write for the user only.

file-type is the file type of the name. The default value of file-type is "tmp".

directory, if supplied, is the directory to create the file in. It defaults to the default temp directory, which is what get-
temp-directory returns, which defaults to what the Operating System uses as the temp directory.

prefix is used as the first part of the file name. The default prefix is "lwtemp_machinename_pid". More characters are appended to make the name unique and random.

If delete-when-close is non-nil, when the stream stream that is returned is closed, the system tries to delete the file quietly. That is, it tries to avoid giving an error if it fails.

element-type and external-format are interpreted the same way as in open.

The stream that is returned is an I/O stream.

The function create-temp-file creates a new temp file and returns the pathname for it. create-temp-file behaves exactly like open-temp-file, as described above, except that it returns a pathname rather than a stream to the new file.

Notes
1. pathname can be called to find the pathname that was used in open-temp-file. The file can be guaranteed to be new only if the temp directory is configured correctly.
2. The default "temp directory" can be found by using get-temp-directory.
3. When delete-when-close is non-nil, it tries to delete the file when the stream is closed, but that does not necessarily succeed. On Microsoft Windows it certainly fails when the file is still opened (for example, by another stream in the same process or another process)

See also
get-temp-directory
open
set-temp-directory
create-universal-binary

Function

Summary

Package hcl

Signature
create-universal-binary target-image src-image1 src-image2 => target-image

Arguments
target-image A pathname designator.
src-image1 A pathname designator.
src-image2 A pathname designator.

Values
target-image A pathname designator.

Description
The function create-universal-binary is deprecated and signals an error.

Compatibility note
In LispWorks 6.1 for Macintosh and earlier versions, create-universal-binary creates a universal binary from two mono-architecture LispWorks images. It must be called only in a LispWorks for Macintosh image that is itself a universal binary.

In LispWorks 7.0 and later versions, universal binaries are not supported and hence create-universal-binary simply signals an error.

See also
save-image
save-universal-from-script

current-function-name

Function

Summary
Return the name of the currently executing function as a string.
The function `current-function-name` returns the name of the currently executing function as a string. `name` is a string representing the name of the function from which `current-function-name` is called. The result is generated by `prin1-to-string` with the variable `*package*` bound to the `KEYWORD` package.

**Notes**

`current-function-name` is for use in debugging, for example to give more context in a run time error message that is produced by a macroexpansion. The result when `current-function-name` is called outside a function (in a Listener or at the top level of a file) is not well defined. It is either `nil` or a name of some internally generated function.

### current-stack-length

**Function**

**Summary**

Returns the size of the current stack.

**Package**

`hcl`

**Signature**

`current-stack-length => stack-size`

**Arguments**

None
Values

stack-size  The current size of the stack, in 32 bit words (in 32-bit implementations) or 64-bit words (in 64-bit implementations).

Compatibility notes

In LispWorks 4.4 and previous on Windows and Linux platforms, \texttt{current-stack-length} was not implemented. This is fixed in LispWorks 5.0 and later.

Example

\begin{verbatim}
(current-stack-length) => 16000
\end{verbatim}

See also

\texttt{extend-current-stack}

\texttt{*sg-default-size*}

date-string  \hspace{1cm} \textit{Function}

Summary  Return a string representing the date and time.

Package  hcl

Signature  \texttt{date-string \textit{optional} seconds expand-month \textit{=>} string}

Arguments

seconds  \texttt{nil} (default) or an integer.

expand-month  A generalized boolean, default false.

Values

string  A string

Description  The function \texttt{date-string} returns a string representing the date and time (including seconds).

If \texttt{seconds} is \texttt{nil} (the default), \texttt{string} represents the current time. Otherwise, \texttt{seconds} is interpreted as a universal time and \texttt{string} represents that time.

If \texttt{expand-month} is true then the date is written as DD MMM YYYY, with the month in characters. Otherwise, the date is written as YYYY/MM/DD, with the month in digits.
The time follows the date, separated by a space, and is always written as HH:MM:SS.

date-string is intended as a quick way of marking some text as related to some time. For example, the function log-bug-form starts by doing something like:

(format stream "=== Log at ~a ===" (date-string))

declaration-information

Function

Summary
Return information about the function bindings of a symbol in an environment.

Package
hcl

Signature
declaration-information decl-name &optional env => info

Arguments
decl-name A declaration name
env An environment or nil

Values
info Information about decl-name

Description
The function declaration-information returns information about the declarations for decl-name in the environment env.

The following values for decl-name are supported:

optimize The value of info is a list of lists of the form (quality value), where quality is one of the optimization qualities specified by the Common Lisp standard and LispWorks extensions (float, for example). Each value is the corresponding value for that quality.

declaration
The value of info is a list of symbols that have been declared as declaration names, for example by use of
\[(\text{declare} \ (\text{declaration} \ ...))\]

There are currently no other supported values for decl-name.

Notes

\texttt{declaration-information} is part of the environment access API which is based on that specified in \textit{Common Lisp: the Language (2nd Edition)}.

See also

\texttt{augment-environment} \newline \texttt{define-declaration} \newline \texttt{function-information} \newline \texttt{map-environment} \newline \texttt{variable-information}

\textbf{*default-package-use-list*} \hspace{1cm} \textit{Variable}

Summary  
List of packages that newly created packages use by default.

Package  
hcl

Initial value  
\("\texttt{CL} \ "\texttt{LW} \ "\texttt{HCL}\"

Description  
This variable is the default value of the :use keyword to \texttt{defpackage}, which specifies which existing packages the package being defined inherits from.

\textbf{*default-profiler-collapse*} \hspace{1cm} \textit{Variable}

Summary  
Controls collapsing of the profile tree.

Package  
hcl
The variable *default-profiler-collapse* is a boolean indicating whether the profile tree should collapse functions with only one child function. The default value is nil.

See also
- print-profile-list
- set-up-profiler

*default-profiler-cutoff*  
Variable

The minimum percentage that the profiler will display in the output tree.

Package  hcl

Initial value  0

The variable *default-profiler-cutoff* is the minimum percentage (0 to 100) that the profiler will display in its output tree. Functions below this percentage will not be displayed. The initial value is 0, meaning display everything.

See also
- print-profile-list
- set-up-profiler

*default-profiler-limit*  
Variable

The maximum number of lines of output that are printed during profiling.

Package  hcl

Initial value  100,000,000
Description  *default-profiler-limit* is the maximum number of lines of output in profile results. The default value is large to ensure that you receive all possible output requested. *default-profiler-limit* only counts output lines for functions that are actually called during profiling. Therefore, if *default-profiler-limit* is 19, and 20 functions were profiled, you would receive full output if one or more of the functions were not actually called during profiling.

See also  
print-profile-list
set-up-profiler

*default-profiler-sort*  
Variable

Summary  The default sorting style for the profiler.

Package  hcl

Initial value  :profile

Description  The variable *default-profiler-limit* controls which column of the profiler’s columnar report is used for sorting. The value can be one of :profile, :call or :top.

See also  
print-profile-list
set-up-profiler

defglobal-parameter  
Function

Summary  Defines a hcl:special-global parameter.

Package  hcl

Signature  defglobal-parameter name initial-value &optional doc => name
Arguments

- **name**: A symbol.
- **initial-value**: A Lisp object.
- **doc**: A string.

Values

- **name**: A symbol.

Description

The macro `defglobal-parameter` has the same semantics as `cl:defparameter`, but also declares the name `name` to be `hcl:special-global`.

See also

- `defglobal-variable`

### defglobal-variable

**Summary**

Defines a `hcl:special-global` variable.

**Package**

`hcl`

**Signature**

`defglobal-variable name &optional initial-value doc => name`

**Arguments**

- **name**: A symbol.
- **initial-value**: A Lisp object.
- **doc**: A string.

**Values**

- **name**: A symbol.

**Description**

The macro `defglobal-variable` has the same semantics as `cl:defvar`, but also declares the name `name` to be `hcl:special-global`.

See also

- `defglobal-parameter`
**define-declaration**

**Macro**

**Summary**
Define a user declaration handler for code walkers.

**Package**
hcl

**Signature**
define-declaration decl-name lambda-list &rest body => decl-name

**Arguments**
- decl-name: A symbol.
- lambda-list: A list of two symbols.
- body: One or more forms.

**Values**
- decl-name: A symbol.

**Description**
The macro **define-declaration** defines a handler for **decl-name**, which tells the compiler and **augment-environment** how to deal with this declaration. The handler is a function with lambda list **lambda-list**, and body **body**, that is the same function as would be produced by:

```
#'(lambda lambda-list . body)
```

When the compiler and **augment-environment** processes a declaration with **decl-name** as the first element, the handler is called with two arguments:

- The declaration itself.
- The environment, which is the compilation environment in the compiler and the new environment in **augment-environment**.

The handler must return two values. The first value specifies what kind of declaration it is, and must be one of:

- **:variable** The declaration applies to variable bindings, and hence affects the result of **variable-information**.
The declaration applies to function bindings, and hence affects the result of \texttt{function-information}.

\textbf{:declare} The declaration does not apply to bindings, and affects the result of \texttt{declaration-information}.

If the first value is \texttt{:variable} or \texttt{:function} then the second value must be a list, the elements of which are lists of the form \texttt{(binding-name key value)}. If the corresponding information function (either \texttt{variable-information} or \texttt{function-information}) is called with \texttt{binding-name} and the environment, then the a-list returned by the information function as its third value will have \texttt{value} associated with \texttt{key}.

If the first value is \texttt{:declare}, then the second value must be a cons of the form \texttt{(key . value)}. The function \texttt{declaration-information} will return \texttt{value} when called with \texttt{key} and the environment.

\texttt{define-declaration} causes \texttt{decl-name} to be proclaimed as a declaration, as if by:

\begin{verbatim}
(proclaim '(declaration decl-name))
\end{verbatim}

\texttt{decl-name} must not be a standard declaration identifier; \texttt{define-declaration} signals an error if it is.

The consequences are undefined if a \texttt{key} returned by a declaration handler defined with \texttt{define-declaration} is a symbol that is used by the corresponding information function to return information about any standard declaration specifier. For example, if the first return value from the handler is \texttt{:variable}, then the second return value should not use the symbols \texttt{dynamic-extent}, \texttt{ignore}, or \texttt{type} as \texttt{key}, because they are reserved by \texttt{variable-information} to return information about the corresponding standard declaration.

\textbf{Notes} Using a declaration defined by \texttt{define-declaration} affects only the return values of \texttt{variable-information}, func-
tion-information or declaration-information as described above. It does not affect the behavior of the compiler. define-declaration is intended for use by code walkers that require extra information in the environment.

The evaluator ignores declarations defined by define-declaration.

define-declaration does not have any compile-time effect so must have be evaluated before a declaration for decl-name is processed.

augment-environment processes declarations last, so the environment that is passed to the handler already contain any other information that was passed to augment-environment.


See also declare
declaration-information
function-information
variable-information
augment-environment
undefine-declaration

delete-advice

Macro

Summary Removes a piece of advice.

Package hcl

Signature delete-advice function-dspec name => nil
Arguments

- **function-dspec**: A function-dspec specifies the function definition to which the piece of advice belongs. See “Function dspecs” on page 84 for description of function-dspec.

- **name**: A symbol naming the piece of advice to be removed. Since several pieces of advice may be attached to a single functional definition, the name is necessary to indicate which one is to be removed.

Values

- **delete-advice** returns `nil`.

Description

- **delete-advice** is used to remove a piece of advice. Advice is a way of altering the behavior of functions. Pieces of advice are associated with a function using `defadvice`. They define additional actions to be performed when the function is invoked, or alternative code to be performed instead of the function, which may or may not access the original definition. As well as being attached to ordinary functions, advice may be attached to methods and to macros (in this case it is in fact associated with the macro’s expansion function).

- **remove-advice** is a function, identical in effect to `delete-advice`, except that you need to quote the arguments.

Notes

- **delete-advice** is an extension to Common Lisp.

See also

- `defadvice`
- `remove-advice`

Chapter 6, “The Advice Facility”

---

delivered-image-p

Function

Summary

The predicate for whether the running image is a delivered image.
## delived-image-p

**Package**
hcl

**Signature**
delivered-image-p => result

**Values**

result

A boolean.

**Description**
The function `delivered-image-p` returns true if the running image is a delivered image, that is an executable or dynamic library created by `deliver`.

Otherwise the running image is a LispWorks development image (potentially a Saved Session or saved explicitly by `save-image`) and then result is false.

**See also**
deliver

---

## deliver-to-android-project

**Function**

**Summary**
Deliver LispWorks for Android. Implemented only in LispWorks for Android Runtime.

**Package**
hcl

**Signature**
deliver-to-android-project function project-path level &key library-name using-ndk no-sub-dir studio-p &allow-other-keys

**Arguments**

function

A symbol.

project-path

A pathname designator.

level

An integer in the inclusive range \([0, 5]\).

library-name

A string.

using-ndk

A boolean.

no-sub-dir

A pathname or a string specifying a directory, or `t` or `nil`.

studio-p

A boolean.
The function `deliver-to-android-project` delivers a LispWorks runtime for the Android platform.

`deliver-to-android-project` creates two files, a Lisp heap and a dynamic library, that needs to be part of an Android project to be used on Android. It does some Android-specific processing, and then calls `deliver`.

If `function` is non-nil it is the restart function which is called after the LispWorks runtime finishes initializing. It is called on another process (by `funcall-async`), and its return value is not used. By the time the function `function` is called, LispWorks is ready to receive calls from Java, and a call from `function` to Java may be used to inform Java that LispWorks is ready instead of the `reporter` argument to `com.lispworks.Manager.init` (or in parallel to it). The function function should return in short time, If you want it to linger, use `process-run-function` to start another with a function that lingers, and return from `function`.

`project-path` is the path of the Android project, except when `no-sub-dir` is supplied, when it defines a directory to put the files. When delivering into an Android Studio project, `project-path` can be either the root of the project or the "main" directory (the location of the file `AndroidManifest.xml`).

`level` is the delivery level. It is passed to `deliver`. See the documentation for `deliver` for details.

`library-name` when supplied must be a string, and defines the base name of the files. The call to `com.lispworks.Manager.init` which initializes LispWorks must match `library-name`. The default value of `library-name` is "LispWorks".

**Note:** Actually the call `com.lispworks.Manager.init` uses the name to find the files, so if you rename the files `com.lispworks.Manager.init` must match the names of the files, rather than `library-name`.
using-ndk needs to be true if the Android project is built using Eclipse and uses NDK to build C code. The default value of using-ndk is nil.

no-sub-dir, when non-nil, tells deliver-to-android-project that the project-path argument is the directory where the files need to go, rather than a project path. In this case deliver-to-android-project does not look for sub-directories. If no-sub-dir is a pathname or string, it specifies a directory where the heap should go. If this directory is relative, it is relative to the project-path directory. If no-sub-dir is t, it specifies that the heap should go at top level in project-path. When no-sub-dir is passed, it is your responsibility to ensure that the files end up in the right place in the Android project. no-sub-dir is useful when the delivery cannot be done directly into the Android Project, for example when it is on a different machine or there are permissions issues. It may also be useful if you have a project with a directory structure that does not match the structure that deliver-to-android-project expects. The default value of no-sub-dir is nil.

studio-p tells deliver-to-android-project whether the target Android project is built with Android Studio or not, which is used to decide where the files go. When studio-p is not passed, deliver-to-android-project tries to determine whether it is Android Studio as described below.

deliver-to-android-project first decides the names of the files to generate and where they should go.

The two files that deliver-to-android-project generates are named lib<library-name>.so.lwheap for the heap file, and lib<library-name>.so for the dynamic library, so by default the names are libLispWorks.so.lwheap and lib-LispWorks.so.

The directory or directories where the files go are determined as follows:
1. When `no-sub-dir` is `nil` (the default), it first checks whether the `project-path` contains a file named `AndroidManifest.xml`.

   a) If the file `AndroidManifest.xml` exists, then this is the "main" path. If `studio-p` was not supplied, it sets `studio-p` to `t` if there is a directory called "java" in the "main" directory, otherwise it sets `studio-p` to `nil`.

   b) If there is no `AndroidManifest.xml` file, it checks whether inside the `project-path` there is a sub-sub-sub-directory `app/src/main`, and whether `AndroidManifest.xml` exists in it. If so, it takes the sub-sub-sub-directory as the "Main" directory, and sets `studio-p` to `t` (ignoring any supplied value).

   Once it decided where the "Main" directory is, it puts the heap in its "assets" sub-directory, and the dynamic library in its "jni libs" sub-directory. The "jni libs" directory is:

   (i) If `studio-p` is `t` it is "jniLibs/armeabi-v7a".

   (ii) Otherwise if `using-ndk` is `nil` it is "libs/armeabi-v7a".

   (iii) Otherwise it is "jni".

2. If `no-sub-dir` is non-nil, then the dynamic library is put in the `project-path` directory. If `no-sub-dir` is `t`, the heap is put in the same place. Otherwise, the directory specified by `no-sub-dir` is merged with the `project-path` to create the path where the heap goes.

   Note: If `no-sub-dir` is a pathname or string it must specify a directory, which can be an absolute path.

After `deliver-to-android-project` determines the names of the files and where they go, it calls `deliver`, passing `function`, the appropriate path, `level`, the Delivery keywords `:split`, `:exe-file`, `:dll-exports` and `:image-type` with the correct values for Android, and all the keyword arguments it was supplied except `library-name`, `using-ndk`, `studio-p` and `no-sub-dir`. The keywords that `deliver-to-android-project`
project passes explicitly should not be used, but the rest of the Deliver keywords can be used and are interpreted in the standard way (see the *LispWorks Delivery User Guide* for details). However, since CAPI is not available on Android, all keywords related to CAPI are not useful.

**Notes**

1. With Android Studio you do not need *using-ndk*. You need it when you use *ndk-build* to build dynamic libraries in your project. In Eclipse that is normally done by the CDT builder. You need *using-ndk* because the *ndk-build* removes all dynamic libraries from the *libs* sub-directory and its sub-directories, including the LispWorks dynamic library. The solution for this is to pass a true value for *using-ndk* to *deliver-to-android-project*, and add to the *Android.mk* file in the *jni* sub-directory the lines that are required to copy the LispWorks dynamic library to *libs/armeabi-v7a*. Assuming you do not pass *library-name*, these lines in *Android.mk* should be:

```
LOCAL_SRC_FILES := libLispWorks.so
LOCAL_MODULE    := LispWorks
include $(PREBUILT_SHARED_LIBRARY)
```

If you pass *library-name*, you need to change the lines above to match your change. There can be other solutions, for example you can deliver elsewhere, and add a build step that copies the LispWorks files into the project after the CDT builder (on Eclipse). If the build in Eclipse is changed not to delete files from the *libs* directory, then *using-ndk* is not needed.

2. Like *deliver, deliver-to-android-project* cannot be called with multiprocessing running, and is best called inside a script that is passed to LispWorks by the command line argument `-build`. 
3. **deliver-to-android-project** is available only in the Android delivery image `lispworks-7-1-0-arm-linux-android`. This image is an ARM image, and must be run on ARM architecture. That can be either an ARM machine, or an ARM emulator.

To deliver a LispWorks for Android Runtime image with a delivery script that calls `deliver-to-android-project` using the QEMU emulator, use the shell script `examples/android/run-lw-android.sh`, which passes its arguments to `lispworks-7-1-0-arm-linux-android`:

```
run-lw-android.sh -build /path/to/delivery-script.lisp
```

4. The algorithm for finding the directories will work for Eclipse and for Android Studio with the default settings. Android Studio is very flexible and still in flux, so may change. It seems unlikely that the names of the sub-directories (`assets`, `jniLibs` and `java`) will change, but the main directory may move around. Supplying the actual main directory rather than the root should still work. If this does not work, you can supply `no-sub-dir` to either put the files in the right directories, or just put it in some directory and copy it to the right place.

See also Chapter 16, “Android interface”

*LispWorks Delivery User Guide*

*disable-trace*  

**Variable**  

<table>
<thead>
<tr>
<th>Summary</th>
<th>Controls tracing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>hcl</td>
</tr>
<tr>
<td>Initial value</td>
<td>nil</td>
</tr>
</tbody>
</table>
**Description**  
*disable-trace* controls tracing without affecting the tracing state. If it is set to `t` then tracing is switched off, but this does not call `untrace`. When the value of *disable-trace* is restored to `nil`, tracing continues as before.

**Notes**  
*disable-trace* is an extension to Common Lisp.

**See also**  
trace

---

**do-profiling**  

*Function*

**Summary**  
A convenience function for profiling multiple threads, combining `start-profiling` and `stop-profiling`.

**Package**  
hcl

**Signature**  
do-profiling &key initialize processes profile-waiting ignore-in-foreign sleep function arguments func-and-args print stream

**Arguments**  
initialize  
A boolean.

processes  
One of :current, :all, a `mp:process` or a list of `mp:process` objects.

profile-waiting  
A boolean.

ignore-in-foreign  
A boolean.

sleep  
A non-negative number, or `nil`.

function  
A function designator.

arguments  
Arguments passed to `function`.

func-and-args  
A function designator or a list `(function-designator . args)`.

print  
A generalized boolean.

stream  
An output stream.
The function **do-profiling** is a convenience function for profiling multiple threads, combining **start-profiling** and **stop-profiling**.

The behavior of **do-profiling** with no arguments is the same as:

```
(progn
  (start-profiling :processes :all)
  (sleep 6)
  (stop-profiling))
```

The arguments **initialize**, **processes**, **profile-waiting** and **ignore-in-foreign** are passed to **start-profiling**. They have the same default values as for **start-profiling**, except **processes** which defaults to **:all**.

The arguments **print** and **stream** are passed to **stop-profiling**. They have the same default values as in **stop-profiling**.

**sleep** is the time to sleep in seconds. If **sleep** is **nil** or 0 **do-profiling** does not sleep. Also, if **sleep** is supplied and either **function** or **func-and-args** are passed, it does not sleep.

**func-and-args**, and **function** together with **arguments**, can both be used for calling a function you supply. **func-and-args** is either a list of the form `(function-designator args)`, in which case **function-designator** is applied to the **args**, or it is a function designator which is called without arguments. **function** is applied to **arguments**.

The order of execution is first **func-and-args** (if this is non-nil), then **function** together with **arguments** if **function** is non-nil, and then sleep if **sleep** was passed explicitly or both **function** and **func-and-args** are **nil**.

**Example**

To profile whatever happens in the next 6 seconds:

```
(hcl:do-profiling)
```

To profile whatever happens in the next 10 minutes:
(hcl:do-profiling :sleep 600)

To run 4 processes in parallel with the same function and profile until they all die:

(defun check-all-processes-died (processes)
  (dolist (p processes t)
    (when (mp:process-alive-p p)
      (return nil))))

(let ((processes
  (loop for x below 4
    collect
      (mp:process-run-function
        (format nil "my process ~a" x)
        () 'my-function))))
  (hcl:do-profiling
    :func-and-args
    (list 'mp:process-wait
      "Waiting for processes to finish"
      'check-all-process-died
      processes)))

See also start-profiling stop-profiling

editor-color-code-coverage

Function

Summary Displays code coverage in an Editor tool for one file.

Package hcl

Signature editor-color-code-coverage filename &key code-coverage-data for-editing show-counters color-covered color-uncovered => result

Arguments

filename A pathname designator.

code-coverage-data A code-coverage-data object.

for-editing A boolean.
The function `editor-color-code-coverage` displays code coverage in an Editor tool in the LispWorks IDE for one file. `filename` must specify a source file, which has code coverage information in the `code-coverage-data`.

If `code-coverage-data` is not supplied, it defaults to the internal code coverage data, that is its binary file with code coverage data was loaded in the current image or `restore-code-coverage-data` was called with data that contains this file. Otherwise, it must specify a `code-coverage-data` object with data for this file.

`for-editing` specifies whether is intended that the buffer with the coloring will be editable. When `for-editing` is `nil`, a buffer without a pathname is created with a different name from the source file, which prevents accidental overwriting of the source file. If `for-editing` is non-nil, the file is opened in the normal way, which may mean using an existing editor buffer if it is already opened. Unless you supply `show-counters`, a buffer that is opened with `for-editing` non-nil does not contain any modification of the source code. The default value of `for-editing` is `nil`.

Depending on the value of `comment-counters`, the counters may be wrapped by `#| |#`. When `show-counters` is non-nil,
counters are inserted inside the source code. The counters are wrapped in #| |#, so the code is still functional, but less readable. The default value of show-counters is (not for-editing).

color-covered and color-uncovered control whether to color covered and uncovered forms respectively. The default value of color-covered is nil. The default value of color-uncovered is t. The default for color-covered and color-uncovered can be set in the LispWorks IDE Preferences... dialog for Code Coverage Browser, tab Coloring.

font-lock-p controls whether font lock (that is, color according to Lisp syntax in the normal way) should be done. When it is t, if the buffer is not already "font locked", it is "font locked" before coloring for code coverage. If font-lock-p is :force, the buffer is always "font locked" first.

color-covered and color-uncovered control whether to comment counters when they are added. It has no effect when show-counters is nil. When the counters are commented, the code is still valid, because the reader just skips the counters, so you can edit and compile it. When the counters are not commented, the code is not valid, but it is easier to read. The default for comment-counters can be set in the Preferences... dialog for Code Coverage Browser, tab Coloring. The initial default value of comment-counters is t.

runtime-only controls whether to display only run time forms, which means exclude forms that execute only at compile time or load time. The default for runtime-only can be set in the Preferences... dialog for Code Coverage Browser, tab Coloring. The initial default value of runtime-only is nil.

real-filename may be used to specify the actual file to load. When it is non-nil, filename should be a pathname which is the same as the truename that the compiler used when it compiled the file to generate the code coverage. The filename is used to lookup the data in the code-coverage-data object, while real-filename is used as the actual text to load. Note that while filename needs to be the same as the truename that the
The compiler used, it is not necessarily a real truename on the current machine.

`editor-color-code-coverage` returns the editor buffer if it is successful. If it fails it returns a list containing a format string followed by format arguments, which can be used to present an error or message to the user.

If `editor-color-code-coverage` succeeds and `for-editing` is `nil`, it remembers that it generated the buffer for the `filename`, and if it is called again with the same `filename` and `for-editing` `nil` and succeeds, deletes the previous buffer.

See “Understanding the code coverage output” on page 124 for details of how to interpret the coloring.

**Notes**

`real-filename` is used when the coloring is done on a machine which sees the file via a different pathname than the machine that compiled it, or when the code coverage data is generated from a copy of the source. The mapping in the Code Coverage browser uses it. Figuring out the truename on a different machine is not always easy. The best way is to use the one from the data, which you can find either by searching the data using `map-code-coverage-data`, or from a `code-coverage-file-stats` object if you already have it.

See also

“Understanding the code coverage output” on page 124
Chapter 10, “Code Coverage”
`code-coverage-data`

### enlarge-generation

**Function**

**Summary**

Enlarges a generation in 32-bit LispWorks.

**Package**

`hcl`

**Signature**

`enlarge-generation gen-num size => result`
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen-num</td>
<td>A generation number.</td>
</tr>
<tr>
<td>size</td>
<td>The amount (in bytes) by which the generation is to be enlarged.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>A boolean.</td>
</tr>
</tbody>
</table>

Description

The function `enlarge-generation` enlarges generation `gen-num` by `size` bytes. If possible, an existing segment in generation `gen-num` is enlarged, otherwise a new segment of size `size` is added to the generation.

`result` is `t` on success and `nil` on failure.

This function is useful when it is known that a generation will need to grow. After `enlarge-generation` is called, the garbage collector is saved the work of deducing that the generation must grow.

`enlarge-generation` is most useful in non-interactive applications, where relatively long GC delays are not a problem. In this case, enlarging generations 0 and 1 by several MB may improve the overall performance of the GC.

Notes

`enlarge-generation` is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations. In 64-bit implementations you can use `set-default-segment-size`.

See also

`set-default-segment-size`

“Memory Management in 32-bit LispWorks” on page 135

enlarge-static

Function

Summary

Enlarges the size of the first static segment in 32-bit LispWorks.

Package

`hcl`
**enlarge-static**

**Signature**

\[ \text{enlarge-static} \ size \Rightarrow result \]

**Arguments**

- **size**
  A non-negative \texttt{fixnum}.

**Values**

- **result**
  A boolean.

**Description**

This function can be used when the system would otherwise allocate additional static segments. Such additional segments would cause the application to grow irreversibly.

- **size** is the amount (in bytes) by which the static segment is to be enlarged. It is rounded up to a multiple of 64K.

- **result** is \texttt{t} if the static segment was successfully enlarged, and \texttt{nil} otherwise.

Use \texttt{room}, with argument \texttt{t}, to find the size of the static segments, and thus the size by which to enlarge the first static segment.

**Notes**

\texttt{enlarge-static} is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations, where the irreversible growth problem described above does not exist.

**See also**

- \texttt{in-static-area}
- \texttt{room}
- \texttt{set-default-segment-size}
- \texttt{switch-static-allocation}
- “Memory Management in 32-bit LispWorks” on page 135

### ensure-hash-entry

**Function**

**Summary**

Gets a value from a \texttt{hash-table}, adding a new value if this fails, all with the table locked.

**Package**

\texttt{hcl}
### Signature

```
ensure-hash-entry key hash-table new-value &optional in-lock-constructor => result
```

### Arguments

- **key**: A Lisp object.
- **hash-table**: A hash-table.
- **new-value**: A Lisp object.
- **in-lock-constructor**: A function designator for a function of one argument.

### Values

- **result**: A Lisp object.

### Description

The function `ensure-hash-entry` gets the value for the key `key` in the hash table `hash-table`, and if this fails puts a new value `new-value` in `hash-table` and returns it. `ensure-hash-entry` does all this with `hash-table` locked.

If the key `key` is not found, then if `in-lock-constructor` is non-nil then `in-lock-constructor` is called with `new-value` as its argument, and the result is put in the table and returned. If `key` is not found and `in-lock-constructor` is `nil`, `new-value` is put in the table and returned.

### Notes

`ensure-hash-entry` is quite inefficient because it always locks the hash table. Normally you should use `with-ensuring-gethash` or `gethash-ensuring` instead.

### See also

- `gethash-ensuring`
- `with-ensuring-gethash`

  “Modifying a hash table with multiprocessing” on page 279

---

**error-situation-forms**

### Macro

### Summary

Informs the compiler of "error situation" forms.
### Package

**hcl**

### Signature

`error-situation-forms &body body => result`

### Arguments

**body**  
Lisp forms.

### Values

**result**  
The result of evaluating `body`.

### Description

The macro `error-situation-forms` tells the compiler that a body of code comprises "error situation" forms.

`body` is evaluated as an implicit progn, but its forms are treated as "error situation" forms. Currently that means that the compiler does not generate code coverage inside `body` or for the `(error-situation-forms ...) form itself, unless `force` was supplied non-nil to `generate-code-coverage` or `with-code-coverage-generation`.

In the future, it may also affect other parameters.

### Notes

For code coverage, `error-situation-forms` differs from `without-code-coverage` in that it does not generate a counter for the `(error-situation-forms ...) form itself, and therefore is more convenient to use.

### Examples

```lisp
(if (check-something)
    (ok-code)
    (error-situation-forms (call-error)))
```

### See also

`without-code-coverage`  
`generate-code-coverage`  
`with-code-coverage-generation`

### Function

**expand-generation-1**

#### Summary

Controls expansion of generation 1 in 32-bit LispWorks.

#### Package

**hcl**
Signature  
\texttt{expand-generation-1 \texttt{on}}

Arguments  
\texttt{on \quad \texttt{t, nil or 1.}}

Description  
The function \texttt{expand-generation-1} controls the subsequent behavior of the garbage collector when insufficient space is freed by a \texttt{mark-and-sweep}. When this occurs, either generation 1 is expanded, or the objects in it are promoted.

If \texttt{on} is \texttt{nil}, generation 1 is never expanded.

If \texttt{on} is \texttt{t}, generation 1 is always expanded (rather than promotion) when needed.

If \texttt{on} is \texttt{1}, generation 1 is only expanded if its current size is less than \texttt{500000} bytes. This is the initial setting.

Notes  
\texttt{expand-generation-1} is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations, where you can use \texttt{set-default-segment-size}.

See also  
\texttt{clean-generation-0}  
\texttt{collect-generation-2}  
\texttt{collect-highest-generation}  
\texttt{mark-and-sweep}  
\texttt{set-default-segment-size}  
\texttt{set-gc-parameters}  
“Memory Management in 32-bit LispWorks” on page 135

\textbf{extend-current-stack}  
\textit{Function}

Summary  
Extends the current stack.

Package  
\texttt{hcl}

Signature  
\texttt{extend-current-stack \&optional how-much => size}
Arguments  

how-much  What percentage the stack should be extended by. The default is 50.

Values  

size  The new size of the stack, after extending.

Description  

Extend the current stack by the given percentage.

Compatibility notes  

In LispWorks 4.4 and previous on Windows and Linux platforms, extend-current-stack is not implemented. This is fixed in LispWorks 5.0 and later.

Example  

To double the size of the current stack:

(hcl:extend-current-stack 100)

See also  

current-stack-length
*stack-overflow-behaviour*

---

**extended-time**  

Macro

Summary  

Prints useful timing information, including information on garbage collection (GC) activity.

Package  

hcl

Signature  

extended-time &body body

Arguments  

body  The forms to be timed.

Description  

The macro extended-time runs the forms in body. It then prints a summary of the time taken followed by a breakdown of time spent in the GC.

The three columns of the GC breakdown show, respectively, total time, user time, and system time, all in seconds. The rows of the GC breakdown indicate the type of activity.

In 32-bit LispWorks these rows begin:
main promote
indicates promotions from generation 0.

internal promote
indicates when an attempt to promote from one generation to the next causes promotion of the higher generation, to make room for the objects from the lower generation.

fixup is a part of the compaction and promotion process.

In 64-bit LispWorks these rows begin:

Standard gen-num (n calls)
indicates n Standard GCs (includes automatic GCs and calls to gc-generation) in which the highest generation collected was gen-num.

Marking gen-num (n calls)
indicates n Marking GCs (includes calls to marking-gc) in which the highest generation collected was gen-num.

Thus in the example below

Standard 1 ( 1 calls) ...
indicates that there was 1 Standard GC in which the highest generation collected was 1.

Notes extended-time does not print Garbage Collector times if it is used while GC timing is on (after start-gc-timing is called, and before get-gc-timing is called with reset non-nil).

Example This example illustrates output in 32-bit LispWorks:
CL-USER 57 > (extended-time (foo))
Timing the evaluation of (PROGN (FOO))
User time = 26.703
System time = 0.109
Elapsed time = 27.047
Allocation = 40021902832 bytes
0 Page faults

/ system
  total gc activity = 3.312500 /
  3.312500 / 0.000000
  main promote ( 1 calls) = 0.000000 /
  0.000000 / 0.000000
  mark and sweep ( 7305 calls) = 3.312500 /
  3.312500 / 0.000000
  internal promote ( 0 calls) = 0.000000 /
  0.000000 / 0.000000
  promote ( 0 calls) = 0.000000 /
  0.000000 / 0.000000
  fixup ( 1 calls) = 0.000000 /
  0.000000 / 0.000000
  compact ( 0 calls) = 0.000000 /
  0.000000 / 0.000000
10006387712

This example illustrates output in 64-bit LispWorks:
CL-USER 3 > (extended-time (foo))
Timing the evaluation of (PROGN (FOO))

User time    =       11.433
System time  =        0.268
Elapsed time =       11.197
Allocation   = 80040251696 bytes
5 Page faults

 total / user /
 system
 total gc activity = 2.168062 / 2.126444
/ 0.041618
 Standard 0 (28545 calls) = 2.153886 / 2.119799
/ 0.034087
 Standard 1 ( 1 calls) = 0.014176 / 0.006645
/ 0.007531
10006387712
0.0

See also
start-gc-timing
time
“Guidance for control of the memory management system” on page 130

fasl-error

Condition

Summary
The class of error signaled when loading a file which is not a proper fasl file.

Package
hcl

Superclasses
simple-error
file-error
stream-error

Description
The condition class fasl-error is used by load and load-data-file to signal an error when the file is not a proper binary file ("fasl file"), or seems to be corrupted.

See also
load-data-file
fast-directory-files
fdf-handle-directory-p
fdf-handle-directory-string
fdf-handle-last-access
fdf-handle-last-modify
fdf-handle-link-p
fdf-handle-size
fdf-handle-writable-p

Functions

Summary
Maps a callback on the names of files in a specified directory and returns a list of those for which the callback returned true. The callback can retrieve information about the files.

Package
hcl

Signature
fast-directory-files dir-pathname callback => result
fdf-handle-directory-p fdf-handle => directory-p
fdf-handle-directory-string fdf-handle => directory-string
fdf-handle-last-access fdf-handle => last-access
fdf-handle-last-modify fdf-handle => last-modify
fdf-handle-link-p fdf-handle => link-p
fdf-handle-size fdf-handle => size
fdf-handle-writable-p fdf-handle => writable-p

Arguments
dir-pathname A pathname designator without wild characters in its directory path.
callback A function designator.
fdf-handle An opaque object used to retrieve information about a file in dir-pathname.

Values
result A list of strings.
directory-p, link-p, writable-p
Booleans.

directory-string  A string.

last-access, last-modify, size  Integers.

**Description**

The function `fast-directory-files` maps the function `callback` on the names of the files in directory specified by `dir-pathname`, and returns a list of the names for which `callback` returned non-nil.

`dir-pathname` must be a pathname designator, which does not contain wild characters in its directory path. To be useful, it should either be a directory (with no name and type), or with wild name and/or type.

`callback` must be a function of two arguments, the name of the file and an opaque object which is referred to as the `fdf-handle`. The `fdf-handle` can be used to retrieve information about the file, by calling any of the `fdf-handle-*` functions documented on this page.

`fast-directory-files` traverses the files that match `dir-pathname` in an undefined way, and for each file calls the `callback` with the file’s name (not including the directory) and a `fdf-handle`. If `callback` returns non-nil it adds the name to a list. It returns the list of names for which the `callback` returned non-nil. Note that the names do not contain the directory name.

The `fdf-handle` can be accessed by the following readers. Functions named in parentheses would return the same value when called on the full path of the file:

`fdf-handle-size` returns the size of the file in bytes.

`fdf-handle-last-modify` returns the universal time of the last modification of the file (`cl:file-write-date`).

`fdf-handle-last-access` returns the universal time of the last access of the file.
fdf-handle-directory-p is a predicate for whether the file is a directory (file-directory-p).

fdf-handle-link-p is a predicate for whether the file is a soft link (always returns nil on Windows).

fdf-handle-writable-p is a predicate for whether the file is writable (file-writable-p).

fdf-handle-directory-string returns a string with the directory path followed by a separator. Therefore the full path of the file can be constructed by:

```
(string-append (fdf-handle-directory-string fdf-handle) name)
```

Notes

The fdf-handle can be used only within the dynamic scope of the callback to which it was passed.

See also
directory
"Fast access to files in a directory" on page 465

**file-string**

*Function*

**Summary**

Returns the contents of a file as a string.

**Package**

hcl

**Signature**

```
file-string file &key length external-format => string
```

**Arguments**

- `file` A pathname, string or file-stream, designating a file.
- `length` The number of characters to return in string, or nil (the default).
- `external-format` An external format specification, default value :default.

**Values**

- `string` A string containing characters from file.
Returns the entire contents of file (if length is nil), or the first length characters, as a string.

CL-USER 26 > file-string "configure.lisp" :length 18
";;; -*- Mode: Lisp;"

See also guess-external-format

file-writable-p

Tests whether a file is writable.

file-writable-p file => result

file A pathname, string or file-stream, designating a file.

result t or nil

The function file-writable-p checks whether file is writable. Note that this checks the properties of the file, so trying to write to the file may still fail if the file is non-writable for other reasons, for example if it is opened for writing by another program.

CL-USER 44 > file-writable-p (sys:lispworks-file "private-patches/load.lisp")
T

filter-code-coverage-data

Filters information from a code-coverage-data object.

hcl
Signature

```lisp
filter-code-coverage-data ccd filter &key without-stats name => result
```

Arguments

- `ccd`: A `code-coverage-data` object or `t`.
- `filter`: A string, a function, or a fbound symbol.
- `without-stats`: A boolean.
- `name`: A Lisp object, normally a symbol or a string.

Values

- `result`: A `code-coverage-data` object.

Description

The function `filter-code-coverage-data` creates a new `code-coverage-data` object with information for some of the files in the argument `ccd`, as determined by the `filter`. If `ccd` is `t`, this is interpreted as the internal `code-coverage-data` object.

If `filter` is a string, it is interpreted as a regexp (see `find-regexp-in-string`) which is applied to the namestring of each file. `without-stats` is ignored in this case.

If `filter` is a function or a fbound symbol, it is applied to the truename of each file and stats object or only to the truename, depending on `without-stats`. If `without-stats` is `nil` the `filter` is applied to the truename and a `code-coverage-file-stats` object for the file. If `without-stats` is true, the `filter` is applied only to the truename. The default value of `without-stats` is `nil`.

`name` is the name supplied to the new `code-coverage-data` object. The default value of `name` is "Filter".

See also

- `code-coverage-data`
- `code-coverage-file-stats`
- `map-code-coverage-data`

Chapter 10, “Code Coverage”
find-object-size  

Function  

Summary  
Returns the size in bytes of the representation of any Lisp object.

Package  
hcl

Signature  
find-object-size object => size

Arguments  
object  
Any Common Lisp form.

Values  
The result is an integer which is the number of bytes of heap memory currently used to represent the object. If the object takes up no heap memory (fixnum or character), then 0 is returned. Such objects are represented by an immediate value held in a single machine “word”.

The size of a heap object includes hidden space required to hold type and other information; for instance, a base-string of 10 1-byte characters occupies more than 10 bytes of memory.

Description  
Certain Common Lisp objects are not represented by a single heap object; for instance, using find-object-size on a hash-table is misleading as the function returns the size of the hash-table descriptor, rather than the total of the descriptor and the hash-table-array. General vectors and arrays also have this property. All symbols are of the same size, since the print name is not part of a symbol object.

Example  
(hcl:find-object-size  
(make-string 1000 :initial-element #\A  
:element-type 'base-char))  
=>  
1012

See also  
room  
total-allocation
**find-throw-tag**  
*Function*

**Summary**  
The predicate for whether there is a specific catch in the dynamic scope.

**Package**  
hcl

**Signature**  
`find-throw-tag tag => result`

**Arguments**  
tag  
A catch tag.

**Values**  
result  
A boolean.

**Description**  
The function `find-throw-tag` is the predicate for whether there is a catch in the dynamic scope with the supplied catch tag `tag`, so that `cl:throw` will succeed to throw to it.

**Notes**  
`find-throw-tag` needs to traverse all the catch frames on the stack until it finds the tag, and therefore would be slower then checking a dynamically bound variable. If the check needs to be called often, then it is normally better to bind a special variable when the catch is established, and then check that variable. In situations when the check is rare (for example, it is called only in cases of error), using `find-throw-tag` is better because it eliminates the overhead of binding the special.

**See also**  
`throw-if-tag-found`

**finish-heavy-allocation**  
*Function*

**Summary**  
 Tells the system that allocation of many long-lived objects is over.

**Package**  
hcl
The function `finish-heavy-allocation` tells the system that the application finished doing 'heavy' allocation, and from that point onwards allocation is 'normal'. The main distinction between heavy and normal allocation is the typical lifetime of objects: normal allocation means most of new objects are ephemeral, while heavy allocation a large proportion of the new objects are long-lived.

Heavy allocation normally happens when loading, either the application itself or large amount of data. Operations that do not involve loading will almost always be normal. Hence the time that is useful to call `finish-heavy-allocation` is after loading something.

See also `with-heavy-allocation`

---

**flag-not-special-free-action**  
*Function*

**Summary**

Unflags an object for special action on garbage collection.

**Package**

`hcl`

**Signature**

`flag-not-special-free-action object => nil`

**Arguments**

`object`  
The object on which the special actions are to be removed.

**Values**

Returns `nil`.

**Example**

```
CL-USER 1 > (make-instance 'capi:title-pane)
#<CAPI:TITLE-PANE "" 20F9898C>

CL-USER 2 > (flag-not-special-free-action *)
NIL
```
See also
add-special-free-action
flag-special-free-action
remove-special-free-action

flag-special-free-action

Function
Summary
Flags an object for special action on garbage collection.

Package
hcl

Signature
flag-special-free-action object => t

Arguments
object
The object on which the special actions are to be performed. This cannot be a symbol.

Values
Returns t.

Description
Note that all the current special-free-action functions are performed on the object. Use flag-not-special-free-action to unflag an object.

Notes
Each object that is flagged for special free action adds some overhead to every garbage collection. This is not significant for a small number of objects, but calling flag-special-free-action with a large number of objects may slow the system significantly. Thus you should avoid using special free actions where possible. Normally, they should be used only for objects that keep some external resources which need to be freed.

Example
CL-USER 29 > (make-instance 'capi:title-pane)
#<CAPI:TITLE-PANE "" 20F9898C>

CL-USER 30 > (flag-special-free-action *)
T
See also

- add-special-free-action
- flag-not-special-free-action
- remove-special-free-action

**function-information**

*Function*

**Summary**
Return information about the function bindings of a symbol in an environment.

**Package**
hcl

**Signature**

```
function-information function-name &optional env => kind, localp, decls
```

**Arguments**

- **function-name** A function name
- **env** An environment or nil

**Values**

- **kind** Either nil, or one of the keywords :special, :lexical, :symbol-macro and :constant.
- **localp** A boolean
- **decls** An a-list

**Description**
The function `function-information` returns information about how `function-name` is bound in the environment `env`. `function-name` can be a symbol or `setf` function name.

The value of `kind` will be as follows:

- **nil** There is no information about variable in `env`
- **:macro** `function-name` has a macro binding in `env`
- **:function** `function-name` has a function binding in `env`
- **:special-form** `function-name` has a special operator binding in `env`
localp will be true if function-name is bound by a form that has indefinite scope (for example \texttt{flet}) or false if function-name has global scope (for example \texttt{defun}).

decls is an a-list of declarations that refer to function-name. The \texttt{cdr} of each pair is specified according to the \texttt{car} of the pair as follows:

dynamic-extent

The \texttt{cdr} is non-nil if function-name is declared \texttt{dynamic-extent} in \texttt{env}.

inline

The \texttt{cdr} is inline or notinline if function-name is explicitly declared \texttt{inline} or \texttt{notinline} in \texttt{env}. The \texttt{cdr} is \texttt{nil} (or the pair is omitted) if this information is not known.

ftype

The \texttt{cdr} is the type specifier that is declared for function-name in \texttt{env} if any.

Notes
1. Not all of these declarations are supported.
2. function-information is part of the environment access API which is based on that specified in Common Lisp: the Language (2nd Edition).

See also

\texttt{augment-environment}
\texttt{declaration-information}
\texttt{define-declaration}
\texttt{map-environment}
\texttt{variable-information}

\textbf{get-code-coverage-delta}
\textbf{reset-code-coverage-snapshot}
\textbf{set-code-coverage-snapshot}

\textbf{Functions}

Summary
Generate "deltas", which are \texttt{code-coverage-data} objects with information for a period.
Package  hcl

Signature  get-code-coverage-delta &key snapshot name => ccd
reset-code-coverage-snapshot => nil
set-code-coverage-snapshot => t

Arguments  snapshot         A boolean.
name          A Lisp object, normally a symbol or a string.

Values  ccd            A code-coverage-data object.

Description  The function get-code-coverage-delta returns a code-
coverage-data object with information covering the period
since the previous snapshot, and with name name. Normally
this would be set by set-code-coverage-snapshot or get-
code-coverage-delta with snapshot non-nil. If there was no
such previous call, then the "delta" period commences, for
each file, from the time it was loaded.

The function reset-code-coverage-snapshot eliminates
any snapshot. This is useful because the snapshot uses mem-
ory.

The function set-code-coverage-snapshot creates a snap-
shot of the internal code coverage data, to be used by get-
code-coverage-delta.

When snapshot is non-nil, get-code-coverage-delta sets
up a new snapshot. This is more efficient than using set-
code-coverage-snapshot again, but otherwise has the
same effect. The default value of snapshot is nil.

Notes  1. The functions reset-code-coverage, clear-code-cov-
erage and restore-code-coverage-data also elimi-
nate the snapshot.
2. Code coverage manipulation functions like `subtract-code-coverage-data` can also be used to compute deltas, but `get-code-coverage-delta` will normally do it using less memory.

See also

- `clear-code-coverage`
- `code-coverage-data`
- `reset-code-coverage`
- `restore-code-coverage-data`
- `subtract-code-coverage-data`

### gc-generation

**Function**

**Summary**

Does a Copying GC.

**Package**

`hcl`

**Signature**

```
gc-generation gen-num &key coalesce promote block => allocation
```

**Arguments**

- `gen-num` An integer between 0 and 7 inclusive, or `t`.
- `coalesce` A generalized boolean.
- `promote` A generalized boolean.
- `block` An integer between 0 and 7, inclusive, or one of the keywords `:blocking-gen-num` and `:all`.

**Values**

- `allocation` The total allocation in generation `gen-num` and younger generations.

**Description**

The function `gc-generation` does a Garbage Collection of a specific generation. The actual operation is different between 64-bit LispWorks and 32-bit LispWorks.

`gen-num` should be a valid generation number, or `t`. The value `t` is mapped to the blocking generation number in 64-bit...
LispWorks, and to 2 in 32-bit LispWorks. For backwards compatibility the keyword :blocking-gen-num is also accepted, with the same meaning as t.

It is especially helpful to GC the blocking generation (or other higher generations) when large, long-lived data structures become garbage. This is because higher generations are rarely collected by default. For the higher generations, the GC takes longer but recovers more space.

Another situation which may require gc-generation is when objects are marked for special free action (by flag-special-free-action or free-function in a weak hash table). If such objects live long enough to be promoted to higher generation, they may not be garbage collected long after there are no pointers to them. If the free action is important, you may need to periodically GC higher generation (typically the blocking generation, by passing gen-num t).

Operation in 64-bit LispWorks

By default gc-generation operates on the live objects in generation gen-num and all lower generations at or above the generation specified by block by copying them inside their current generation, and it operates on the live objects in generations lower than block by copying them to the next higher generation.

If promote is non-nil, the live objects in generation gen-num are also promoted to the next generation. That is the same operation that happens when the GC is invoked automatically. The default value of promote is nil.

If coalesce is non-nil, all non-static live objects in lower generations are promoted to generation gen-num. That is what clean-down does (with gen-num being the highest generation). It may be useful directly in some cases. The default value of coalesce is nil.
block specifies a generation number up to which to promote. An integer value specifies the generation number. If block is :blocking-gen-num, then gc-generation promotes up to the blocking generation. If block is :all, then gc-generation promotes nothing. The default value of block is :blocking-gen-num.

gc-generation is useful when you know points in your application where many objects tend to die, or when you know that that application is less heavily loaded at some time. Typically many objects die in the end (or beginning) of an iteration in a top level loop of the application, and that is normally a useful place to put a call to gc-generation of generation 2 or generation 3. If you know a time when the application can spend time garbage collecting, a call to gc-generation with a higher value of gen-num may be useful. It is probably never really useful to use gc-generation on generation 0 or 1.

To decide on which gen-num to call gc-generation, check which generation gets full by making periodic calls to room. gc-generation with promote or coalesce may also be useful to move objects from the blocking generation to higher generations, which does not happen automatically (except when saving the image). For example, after loading a large amount of code, and before generating any data that may die shortly, assuming the blocking generation is 3, it may be useful to do:

(gc-generation 4 :coalesce t)

to move all (non-static) objects to generation 4, where they will not be touched by the GC any more (except following pointers to younger generations).

**Operation in 32-bit LispWorks**

gc-generation marks and sweeps the generation gen-num and all generations below, and then does some additional cleanups. coalesce, promote and block are ignored.
Operation in the Mobile GC

When the \textit{gen-num} argument is a number, it must be 0, 1 or 2. The value \texttt{t} (and \texttt{:blocking-gen-num}) is interpreted as 2.

Generation 0 is always promoted, but the \texttt{:promote} keyword affects generation 1 and, if non-nil, promotes even if promotion was blocked by \texttt{set-promote-generation-1}.

The keyword \texttt{:block} is ignored.

Otherwise, the function acts as in 64-bit LispWorks above.

Compatibility notes

In 32-bit LispWorks, \texttt{gc-generation} simply calls \texttt{mark-and-sweep}. This has a similar effect, but two significant differences must be noted:

1. by default, \texttt{gc-generation} promotes the young generations, so repeated calls to \texttt{gc-generation} will promote everything to generation \textit{gen-num} or generation \textit{block} (whichever is lower). In contrast \texttt{mark-and-sweep} never promotes.

2. In 32-bit LispWorks, generation 2 is the blocking generation. In 64-bit LispWorks, the default blocking generation is generation 3. That is because the 64-bit implementation promotes faster and so needs more generations before the block.

Also note that

\begin{verbatim}
(gc-generation t)
\end{verbatim}

is intended as the replacement of

\begin{verbatim}
(mark-and-sweep 2)
\end{verbatim}

See also

\begin{verbatim}
clean-down
mark-and-sweep
marking-gc
set-blocking-gen-num
"Guidance for control of the memory management system"
\end{verbatim}
on page 130
**gc-if-needed**

**Function**

**Summary** Garbage collects if the previous call requires more space that is actually available in 32-bit LispWorks.

**Package** hcl

**Signature** gc-if-needed => nil

**Arguments** None.

**Values** Returns nil.

**Description** This function checks to see if the amount of allocation from the previous call is more than system:*allocation-interval*, and if it is, performs a mark and sweep and promotion on generation 0. It also tries to reduce the big-chunk area. This is a fairly brief operation, and can be used whenever some operation is finished and may have left some garbage. The system itself uses it after compiling and loading files, when waiting for input, etc.

**Notes** gc-if-needed does nothing in 64-bit LispWorks.

**See also**
- avoid-gc
- get-gc-parameters
- mark-and-sweep
- normal-gc
- set-gc-parameters
- without-interrupts
- with-heavy-allocation

“Memory Management in 32-bit LispWorks” on page 135

**generate-code-coverage**

**Function**

**Summary** Switches code coverage generation on or off.
The function `generate-code-coverage` switches code coverage generation on or off.

`on` determines whether code coverage is generated. If `on` is true, code coverage generation is switched on, which means that when `compile-file` is called in the conventional way, that is generate a binary file from a source file, it generates code coverage code. If `on` is `nil`, code coverage generation is switched off and in this case the other keyword arguments are ignored. The default value of `on` is `t`.

`generate-code-coverage` returns `t` or `nil`, depending on the value of `on`.

`atomic-p` controls whether counting is done atomically or not. It is ignored when `counters` is `nil`. Passing `atomic-p` true makes the counters atomic, which may be much slower than counting non-atomically, but guarantees that the code is not going to drop counts when running multiprocessing. The default value of `atomic-p` is `nil`.

`counters` controls whether the code coverage code actually counts executions, or simply sets a flag to indicate that the code has been executed. Passing `counters` `nil` generates code
which is a little smaller and faster, but does not count the number of times a piece of code has been executed. The default value of \texttt{counters} is \texttt{t}.

\texttt{force}, if true, forces generating counters in code that is marked not to generate counters by \texttt{without-code-coverage} or \texttt{error-situation-forms}. The default value of \texttt{force} is \texttt{nil}.

\texttt{count-implicit-branch} controls whether to generate counters for implicit branches. Implicit branches are generated by macros like \texttt{cl:when}, where the source only contains the "then" branch, and the "else" branch (which returns \texttt{nil}) is implicit. The other macros are \texttt{cl:unless} (when it is an implicit "then"), and the switch macros \texttt{cl:cond}, \texttt{cl:case} and \texttt{cl:typecase} when they do not have a \texttt{t} or \texttt{otherwise} clause. When \texttt{count-implicit-branch} is true, the compiler generates a counter for the implicit branch, which counts the number of times that the implicit branch was executed. In other words for \texttt{cl:when} this is the number of times that the condition returned \texttt{nil}; for \texttt{cl:unless} this is the number of times that the condition returned true, and for the switch macros it is the number of times that all the clauses returned \texttt{nil}.

When coloring with an implicit branch with counter 0 inside a form with a non-zero counter, there is nowhere to put the color for the uncovered code, so the form is colored as a \texttt{hidden-partial} form (see “Understanding the code coverage output” on page 124).

The default value of \texttt{count-implicit-branch} is \texttt{t}.

\textbf{Notes}

If \texttt{generate-code-coverage} is called outside the body of \texttt{with-code-coverage-generation}, it switches the generation globally. Inside the body of \texttt{with-code-coverage-generation} it switches the generation within the scope of the surrounding \texttt{with-code-coverage-generation}, but has no effect once this \texttt{with-code-coverage-generation} exited.
get-default-generation  

*Function*

**Summary**

Returns the current default generation.

**Package**

`hcl`

**Signature**

`get-default-generation => default-gen`

**Arguments**

None.

**Values**

Returns the current default.

**Description**

By default, all new objects are allocated to a specific generation. This function returns the current value of this default generation.

**Notes**

In 64-bit LispWorks, `get-default-generation` returns 0.

**See also**

`allocation-in-gen-num`

`clean-generation-0`

`collect-generation-2`

`collect-highest-generation`

`expand-generation-1`

`set-default-generation`

`*symbol-alloc-gen-num*`

“Memory Management in 32-bit LispWorks” on page 135
get-gc-parameters  

Function

Summary
Returns the current values of various garbage collector parameters in 32-bit LispWorks.

Package
hcl

Signature
get-gc-parameters parameters => values

Arguments
parameters
A keyword representing a single GC parameter. Any other value means all parameters.

Values
values
If parameters specifies a single GC parameter, the value of that parameter is returned. Otherwise values is an alist containing every GC parameter, together with its current value.

Description
See set-gc-parameters for a full description of these parameters.

With keyword argument, of one of the parameters, the corresponding value is returned.

Notes
get-gc-parameters is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations.
Example

CL-USER 1 > (get-gc-parameters :minimum-overflow)
500000

CL-USER 2 > (pprint (get-gc-parameters t))

((:ENLARGE-BY-SEGMENTS . 10)
 (:MINIMUM-FOR-PROMOTE . 1000)
 (:MAXIMUM-OVERFLOW . 1000000)
 (:MINIMUM-OVERFLOW . 500000)
 (:MINIMUM-BUFFER-SIZE . 200)
 (:NEW-GENERATION-SIZE . 262144)
 (:PROMOTE-MAX-BUFFER . 100000)
 (:PROMOTE-MIN-BUFFER . 200)
 (:MAXIMUM-BUFFER-SIZE . 131072)
 (:MINIMUM-FOR-SWEEP . 8000)
 (:BIG-OBJECT . 131072))

See also

set-gc-parameters

“Memory Management in 32-bit LispWorks” on page 135

get-temp-directory

Function

Summary
Returns a directory that can be used for temporary files.

Package
hcl

Signature
get-temp-directory => directory

Values
directory  A pathname

Description
The function get-temp-directory returns a directory which is likely to be writable and can be used for temporary files.

Notes
By default, the functions create-temp-file and open-temp-file use the result of get-temp-directory as the directory to create their temp file in.
See also  
create-temp-file
example-compile-file
open-temp-file

get-working-directory

Summary  
Finds the current working directory.

Package  
hcl

Signature  
get-working-directory => cwd

Arguments  
None.

Values  
cwd  
The current working directory, as a pathname.

Description  
This function is used to find the current working directory. It returns a pathname, the directory component of which is the current working directory.

Example  
CL-USER 1 > (get-working-directory)
#P"/u/dubya/"

See also  
cd
change-directory

gethash-ensuring

Summary  
A thread-safe way to get a value from a hash-table, adding a value if the key is not already present.

Package  
hcl
**Signature**

```
gethash-ensuring key hash-table constructor &optional in-lock-constructor => result
```

**Arguments**

- **key** A Lisp object.
- **hash-table** A hash-table.
- **constructor** A function designator for a function of no arguments.
- **in-lock-constructor** A function designator for a function of one argument.

**Values**

- **result** A Lisp object.

**Description**

The function `gethash-ensuring` gets the value for the key `key` from the hash table `hash-table`, and if this fails constructs a new value, puts it in the table and returns it. `gethash-ensuring` does this in a thread-safe way, which means that all threads calling it with the same `key` and `hash-table` return the same value (as long as nothing removes it from the table).

If `key` is not found and `constructor` is non-nil, `constructor` is called to construct the new value. `constructor` is called without any lock, and can do whatever is needed. The value that `constructor` returns may be discarded by `gethash-ensuring` if, by the time it returns, there is already a matching value in `hash-table` (added by another thread or even inside `constructor`).

If `in-lock-constructor` is non-nil it is called with the result of `constructor`, or with `nil` if `constructor` is `nil`. `in-lock-constructor` is called with `hash-table` locked, and its return value is guaranteed to be put in the table and to be returned by `gethash-ensuring`. If `in-lock-constructor` is `nil` then the value that is returned by `constructor`, or `nil`, is used.

**Notes**

1. If either the `constructor` or the `in-lock-constructor` is not simple, it is easier to use `with-ensuring-gethash`.

953
2. In most situations, using constructor to do all the work (which requires minimal holding of the lock) is better than using in-lock-constructor. It means that sometimes the work that constructor did is wasted, because another thread put the value in the table, but that overhead is normally less significant than the overhead of holding the lock for longer, with the associated potential deadlocks. Use in-lock-constructor only if it is essential that the result goes into the table.

See also

- ensure-hash-entry
- with-ensuring-gethash

“Modifying a hash table with multiprocessing” on page 279

**Variable**

*handle-existing-defpackage*

**Summary**

Controls LispWorks’ response when defpackage is used on an existing package that is different from the definition given.

**Package**

hcl

**Initial value**

(:warn :modify)

**Description**

The standard explicitly declines to define what defpackage does if the named package already exists and is in a different state to that described by the defpackage form. The variable *handle-existing-defpackage* is an extension to Common Lisp which allows you to select between alternative behaviors that are known to be useful.

The two alternatives are to modify the package to conform exactly to the definition, removing features if necessary, or to merely add features specified in the defpackage but missing from the package. You can also control whether a condition is signaled.

The variable consists of a list of any of the following:
:error    Signal an error.
:warn     Signal a warning.
:add      Add the new symbols to the externals, imports, and so on.
:modify   Modify the package to have only these externals.
:verbose  The signaled errors or warnings also contain details of the differences.

The options :error and :warn cannot be specified at the same time. One of :add and :modify must be specified. Undistinguished internals (that is, internal symbols that are not imported or shadowed), :intern options and sizes are ignored when deciding whether to signal.

Note that when you use :modify some symbols can be uninterned if defpackage imports another symbol with the same name from another package through :import-from, :shadows-import-from or :export. This happens whether the symbol has a definition as a function, a variable, or any other Lisp construct, so after making such a change in the package, you should re-execute the definitions that were (presumably erroneously) attached to the uninterned symbols.

Notes *handle-existing-defpackage* is an extension to Common Lisp.

See also defpackage

*handle-old-in-package*  Variable

Summary Controls the handling of CLtL1-style in-package forms.

Package hcl
The variable \texttt{*handle-old-in-package*} controls what happens when a CLtL1-style \texttt{in-package} form is processed. This refers to the specification in Common Lisp the Language, first Edition, which preceded ANSI Common Lisp and specified \texttt{in-package} as a function with keyword arguments.

The allowed values are as follows:

- \texttt{:quiet} Quietly use the CLtL1 definition of the \texttt{in-package} function.
- \texttt{:warn} Signal a warning and use the old definition.
- \texttt{:error} Signal a continuable error.

See also \texttt{*handle-old-in-package-used-as-make-package*}

\texttt{*handle-old-in-package-used-as-make-package*} Variable

Summary Controls the handling of CLtL1-style \texttt{in-package} forms.

Package \texttt{hcl}

Initial value \texttt{:quiet}

Description The variable \texttt{*handle-old-in-package-used-as-make-package*} controls what happens when a CLtL1-style \texttt{in-package} form which attempts to create a package is processed. This refers to the specification in Common Lisp the Language, first Edition, which preceded ANSI Common Lisp and specified \texttt{in-package} as a function with keyword arguments.

The allowed values are as follows:

- \texttt{:quiet} Handle according to the value of \texttt{*handle-old-in-package*}. 
:warn Signal a warning and create the package.
:error Signal a continuable error.

See also *handle-old-in-package*

### hash-table-weak-kind

**Function**

**Summary** Returns the weak kind of a hash table.

**Signature**

```lisp
hash-table-weak-kind hash-table => weakness-state
```

**Arguments**

- `hash-table` A hash table.

**Values**

- `weakness-state` A keyword or nil.

**Description**

The function `hash-table-weak-kind` returns the weak kind (or weakness state) of the hash table `hash-table`.

See `set-hash-table-weak` for the meaning of the different values of `weakness-state`.

See also `set-hash-table-weak` `make-hash-table`

### load-data-file

**Function**

**Summary** Loads a binary data file created by `dump-forms-to-file` or `with-output-to-fasl-file`.

**Package**
hcl

**Signature**

```lisp
load-data-file pathname &rest load-args &key eval allow-any-type callback => result
```
Arguments

**pathname**
A pathname designator naming a file which must have been created by `dump-forms-to-file` or `with-output-to-fasl-file`.

**load-args**
Arguments which are passed to `load`, after removing the keyword-value pairs for `:allow-any-type` and `:callback` (if supplied).

**allow-any-type**
A generalized boolean. When `allow-any-type` is true and the supplied `pathname` has a type, `load-data-file` tries to load it as a binary file without checking whether the type is known. When `allow-any-type` is `nil`, `load-data-type` tries to load only pathnames with known binary types (that is, either `*binary-file-type*` or in the list `*binary-file-types*`), exactly like `load`. The default value of `allow-any-type` is `t`.

**callback**
A function of one argument which is called with the result of the evaluation of each form in the file (or the form itself if `eval` is `nil`). When `callback` is supplied, the keyword `:print` (which normally would be processed by `load`) has no effect.

**eval**
A generalized boolean which controls whether the form is actually evaluated. When it is `nil`, the form as loaded from the file (without evaluation) is passed to the `callback` (if supplied) and printed (if `:print t` is supplied). When `eval` is non-nil, the form is evaluated before being passed to the `callback` and/or printed. The default value of `eval` is `t`.
Values

Result

A generalized boolean.

Description

The function `load-data-file` loads a fasl file created by `dump-forms-to-file` or `with-output-to-fasl-file`.

`load-data-file` has similar semantics to `load`, but treats fasl files differently:

- It cannot load a fasl generated by `compile-file`.
- It allows loading of fasls generated by `dump-forms-to-file` or `with-output-to-fasl-file`, including those generated by a previous version of LispWorks, or other architectures of LispWorks, provided they have the same byte order.
- It allows the option of a callback that is called on the result of loading the file.

Fasl files generated by `dump-forms-to-file` or `with-output-to-fasl-file` must only be loaded using `load-data-file`.

`load-data-file` never loads a file as a text file, only files that are recognized as binary, which can be one of these possibilities:

- `pathname` has a known type (either `*binary-file-type*` or in the list `*binary-file-types*`), or
- `pathname` has an unknown type and `allow-any-type` is non-nil, or
- `pathname` does not have a type and a matching file with the type matching `*binary-file-type*` is found.

If `load-data-file` ends up trying to load a file that is not a proper binary file, it signals an error of type `fasl-error`.

During the load, each form is loaded and, if `eval` is true, evaluated. If there is a `callback`, it is called with the result of the evaluation. Otherwise, the result may be printed if `:print t` was passed, and is then discarded.
Notes

1. The default value of \texttt{eval} is \texttt{t} to give the same behavior as in LispWorks 6.1 and earlier versions. Passing \texttt{eval} as \texttt{nil} and using a callback is probably a better way of transferring data around, because it avoids the calls to \texttt{eval}. If needed, \texttt{callback} can call \texttt{eval} explicitly.

2. All x86/x64 and ARM architectures have the same byte order, so \texttt{load-data-file} on any x86/x64 or ARM architecture can be used to load a data file that was generated on any x86/x64 or ARM architecture. The reverse byte order is used by Power architecture (IBM AIX and old PowerPC Mac OS X) and SPARC (old Solaris).

3. \texttt{load-data-file} returns the same value as \texttt{load}. In particular, the return value has nothing to do with the forms in the file. To actually have an effect, either the forms themselves have side effects, or the \texttt{callback} argument is used to perform any required side effects.

4. \texttt{load-data-file} does not do any read operation, but if the forms in the file contain symbols (except \texttt{nil}) such symbols need to be interned.

Compatibility notes

1. In LispWorks 6.1 and earlier versions \texttt{load-data-file} was in the \texttt{SYSTEM} package. It is still exported from \texttt{SYSTEM} for backwards compatibility.

2. In LispWorks 6.1 and earlier versions \texttt{load-data-file} gave errors if the type was not recognized, but now by default it allows any type.

3. In LispWorks 6.1 and earlier versions \texttt{load-data-file}, when given a plain lisp file, would load it the same way that \texttt{load} does. In LispWorks 7.0 or later it signals an error of type \texttt{fasl-error}.

4. \texttt{callback} works only when the fasl file was generated by LispWorks 7.0 or later.

Examples

For a simple example see \texttt{dump-forms-to-file}.
See also

dump-forms-to-file
with-output-to-fasl-file
fasl-error
*binary-file-type*
*binary-file-types*
“Transferring large amounts of data” on page 476

*load-fasl-or-lisp-file*

Variable

Summary

Controls the behavior of load for untyped pathnames.

Package

hcl

Description

The variable *load-fasl-or-lisp-file* determines whether (load "foo") should load the binary file (foo.ofasl, foo.ufasl, foo.xfasl etc, depending on platform) or foo.lisp, when both exist. It may take the following values:

:load-newer  If the fasl is out-of-date, the lisp file is loaded, and a warning message is output in verbose mode.

:load-newer-no-warn  Like :load-newer, but without the warning.

:load-fasl  Always choose fasl files in preference to lisp files, but when verbose, warn if the lisp file is newer.

:load-fasl-no-warn  Like :load-fasl, but without the warning.

:load-lisp  Always choose lisp files in preference to fasl.

:recompile  If the fasl file is out-of-date or there is none, compile and load the new fasl.

:maybe-recompile
If the fasl is out-of-date, queries whether to load it, recompile and then load it, or load the lisp file.

Initial value: `:load-fasl`

### make-ring

**Summary**

Creates a "ring" object.

**Package**

hcl

**Signature**

`make-ring size name &optional delete-function => ring`

**Arguments**

- `size`: A positive fixnum.
- `name`: A string.
- `delete-function`: A function designator for a function of one argument.

**Values**

- `ring`: A "ring" object.

**Description**

The function `make-ring` creates a "ring" object, which can hold up to `size` elements. A ring has stack like behavior but is limited in size, and can be rotated.

`size` is the maximum number of elements that the ring `ring` can hold. Once `ring` has this number of elements, if an element is added to `ring` (by `ring-push`), an element is first removed from the ring.

`name` simply names the ring, but has no effect on its functionality. It is used when printing the ring object, and is returned by `ring-name`.

`delete-function`, if supplied, is called each time an element is removed from the ring (by `ring-push`) because it is full. The default value of `delete-function` is `#'identity`. 
The ring keeps the elements in a logical ring with an "insertion position". The function \texttt{ring-push} adds an element before the insertion position. If the ring is full, it first removes the element immediately after the insertion position.

The function \texttt{ring-pop} removes from the ring the element before the insertion point, and returns that element. Thus when using \texttt{ring-push} and \texttt{ring-pop} on their own, the ring behaves like a stack with limited length.

\texttt{rotate-ring} can be used to move the insertion point. \texttt{ring-ref} can be used to index into the ring. \texttt{map-ring}, \texttt{position-in-ring}, and \texttt{position-in-ring-forward} can be used to iterate through the ring's elements.

All access to a ring is thread-safe. Therefore access to a ring may hang if another process keeps it locked. If you need to guarantee no hanging, you can use \texttt{with-ring-locked} with non-nil \texttt{timeout} around the critical calls.

See also  
\texttt{ring-push}  
\texttt{ring-pop}  
\texttt{rotate-ring}  
\texttt{ring-ref}  
\texttt{ring-length}  
\texttt{ringp}  
\texttt{ring-name}  
\texttt{map-ring}  
\texttt{position-in-ring}  
\texttt{with-ring-locked}
make-unlocked-queue
unlocked-queue-read
unlocked-queue.peek
unlocked-queue-ready
unlocked-queue-send
unlocked-queue-count
unlocked-queue-size

Functions

Summary  Create and use an unlocked-queue object.

Package  hcl

Signature  make-unlocked-queue &key size name => unlocked-queue
unlocked-queue-read unlocked-queue => object
unlocked-queue.peek unlocked-queue => object
unlocked-queue-ready unlocked-queue => result
unlocked-queue-send unlocked-queue object => object
unlocked-queue-count unlocked-queue => count
unlocked-queue-size unlocked-queue object => size

Arguments  size  A positive integer.
name  A Lisp object.
unlocked-queue  An unlocked-queue object.
object  A Lisp object.

Values  count  A positive integer.
object  A Lisp object.
result  A boolean.
size  A positive integer.
The function **make-unlocked-queue** creates a new, empty **unlocked-queue** object.


**size** is a hint of the maximum number of objects that are expected to be in the queue simultaneously. The queue is extended as needed, so **size** does not have to be a good guess.

**name** is used when printing the **unlocked-queue** and so it is useful for debugging. **name** is not used otherwise.

**unlocked-queue-read** checks whether there is anything in the queue, and if so removes the first object in the queue and returns it. Otherwise it returns **nil**.

**unlocked-queue-peek** checks whether there is anything in the queue, and if so returns the first object in the queue without modifying the queue. Otherwise it returns **nil**.

**unlocked-queue-ready** returns a boolean specifying whether there is anything in the queue.

**unlocked-queue-send** adds **object** to the end of the queue, extending the queue if needed. It returns its second argument.

**unlocked-queue-count** returns the number of objects in the queue.

**unlocked-queue-size** returns the current size of the queue. Note that it is increased when needed by **unlocked-queue-send**

**See also**

**make-mailbox**

**unlocked-queue**
map-code-coverage-data

Summary Calls a function on each of the files in a code-coverage-data object.

Package hcl

Signature map-code-coverage-data ccd function &key without-stats collect => list

Arguments ccd A code-coverage-data object or t.
function A function designator.
without-stats A generalized boolean.
collect nil, t or :truenames

Values list nil or a list either of truenames or of code-coverage-file-stats objects.

Description The function map-code-coverage-data maps function over the files in ccd and optionally collects items for some of them. If ccd is t, this is interpreted as the internal code-coverage-data object.

The arguments passed to function depend on without-stats. If without-stats is false then function is called with the truename and a code-coverage-file-stats object for the file. If without-stats is true, then function is applied only to the truename. The default value of without-stats is false.

If collect is t (the default), then map-code-coverage-data collects the stats (when without-stats is false) or the truename (when without-stats is true) for each call to function that returns true. If collect is :truenames, then map-code-coverage-data collects the truename for each call to function that returns true.

When collect is nil, map-code-coverage-data returns nil. Otherwise, it returns a list of the objects it collected.
map-ring

Summary
Calls a function on each element of a ring, modifying the element.

Package
hcl

Signature
map-ring ring function

Arguments
ring A ring object created by make-ring.
function A function designator for a function of one argument.

Description
The function map-ring funcalls the function function on each element in the ring in turn, and sets that ring element to the result.

Notes
1. function is called with the ring locked.
2. If you do not intend to modify the elements of ring, ensure that function returns its argument.

See also
make-ring
position-in-ring
position-in-ring-forward

mark-and-sweep

Summary
Garbage collects a specified generation in 32-bit LispWorks.
This function is deprecated: use gc-generation instead.
The HCL Package

Package hcl

Signature mark-and-sweep gen-number => bytes

Arguments gen-number 0 for the most recent generation, 1 for the most recent two generations, and so on up to a maximum (usually 3). Numbers outside this range signal an error.

Values bytes The number of bytes allocated in that generation.

Description mark-and-sweep is used to garbage-collect a specified generation of memory (and all lower generations). A call to this function forces the garbage collector to scan the specified generations. This can be of use in obtaining consistent timings of programs that require memory allocation. Alternatively, performance can sometimes be improved by forcing a garbage collection, when it is known that little memory has been allocated since a previous collection, rather than waiting for a later, more extensive collection. For example, the function could be called outside a loop that allocates a small amount of memory.

It is specially helpful to mark and sweep generation 2 when large, long-lived data structures become garbage, because by default it is never marked and swept. The higher the generation number the more time the mark-and-sweep takes, but also the more space recovered.

Notes mark-and-sweep is implemented only in 32-bit LispWorks, and is deprecated. Use gc-generation instead.

mark-and-sweep is not relevant to the Memory Management API in 64-bit implementations. In 64-bit implementations you can use gc-generation or marking-gc.

Examples (mark-and-sweep 0) ; collect most recent generation (mark-and-sweep 3) ; collect all generations
See also  
avoid-gc  
block-promotion  
get-gc-parameters  
gc-generation  
gc-if-needed  
normal-gc  
set-array-weak  
set-gc-parameters  
set-hash-table-weak  
without-interrupts  
with-heavy-allocation  
“Guidance for control of the memory management system” on page 130

**max-trace-indent**  

**Variable**

**Summary**  
The maximum level of indentation used in trace output.

**Package**  
hcl

**Initial value**  
50

**Description**  
*max-trace-indent* is the maximum indentation that is used during output from tracing. Typically each successive invocation of tracing causes the output to be further indented, making it easier to see how the calls are nested. The value of *max-trace-indent* should be an integer.

**Example**  
USER 8 > (setq hcl:*max-trace-indent* 4)  
4  
USER 9 > (defun sum (n res) (if (= n 0)  
res  
(+ n (sum (1- n) res))))

SUM

USER 10 > (trace sum)  
SUM
merge-code-coverage-data
destructive-merge-code-coverage-data

Summary
Merge two code-coverage-data objects.

Package
hcl

Signature
merge-code-coverage-data ccd1 ccd2 name => result
destructive-merge-code-coverage-data ccd1 ccd2 => ccd1

Arguments
ccd1 A code-coverage-data object or (for merge-code-coverage-data only) t.
ccd2 A code-coverage-data object or t.
name A Lisp object, normally a symbol or a string.

Values
result A code-coverage-data object.

Description
The function merge-code-coverage-data and destructive-merge-code-coverage-data merge two code-coverage-data objects.

Notes
*max-trace-indent* is an extension to Common Lisp.

See also
trace
Merging means taking all the files from \textit{ccd1} together with those files from \textit{ccd2} which do not have information in \textit{ccd1}. For files that appear in both \textit{ccd1} and \textit{ccd2}, the information in \textit{ccd2} is ignored.

\texttt{merge-code-coverage-data} creates a new \texttt{code-coverage-data} object containing the information for each file, and with name \textit{name}.

\texttt{destructive-merge-code-coverage-data} adds to \textit{ccd1} those files from \textit{ccd2} which are not already there, and returns \textit{ccd1}.

If either of the datas is the internal \texttt{code-coverage-data} object, the file information is copied, so it does not change anymore. Otherwise it just copies the pointer, because the file information is read-only.

See also Chapter 10, “Code Coverage”

\texttt{code-coverage-data}

\textbf{modify-hash} \hfill \textit{Function}

Summary \hspace{1em} Reads and writes an entry in a hash table atomically.

Package \hspace{1em} \texttt{hcl}

Signature \hspace{1em} \texttt{modify-hash hash-table key function => new-value, key}

Arguments \hspace{1em} \begin{itemize} \item \texttt{hash-table} \hspace{1em} A hash table. \item \texttt{key} \hspace{1em} An object. \item \texttt{function} \hspace{1em} A function designator. \end{itemize}

Values \hspace{1em} \begin{itemize} \item \texttt{new-value} \hspace{1em} An object. \item \texttt{key} \hspace{1em} An object. \end{itemize}
The function `modify-hash` locks the hash table `hash-table`. It then calls the function `function` with three arguments: `key`, the value currently associated with `key` in `hash-table` (if any), and a flag which is true if the key was in the table. (This last argument is needed in case the associated value is `nil`).

`modify-hash` then sets the result of the function `function` as the value for `key` in the table. `modify-hash` returns two values, the `new-value` and the `key`.

The overall effect is like:

```lisp
(with-hash-table-locked hash-table
  (multiple-value-bind (value found-p)
      (gethash key hash-table)
    (let ((new-value (funcall function key value found-p)))
      (setf (gethash key hash-table) new-value)
      (values new-value key))))
```

but `modify-hash` should be more efficient.

It is guaranteed that no other thread can modify the value associated with `key` until `modify-hash` returns.

`function` is called with `hash-table` locked, so it should not do anything that may require hanging the modification, or that waits for another process that tries to modify the table.

See also `make-hash-table` and `with-hash-table-locked`

“Atomicity and thread-safety of the LispWorks implementation” on page 265

“Modifying a hash table with multiprocessing” on page 279

**normal-gc**

**Function**

**Summary** Returns the image to normal garbage collection activity in 32-bit LispWorks.
Package hcl

Signature normal-gc => t

Arguments None.

Values The function returns the single result t.

Description normal-gc resets various internal parameters that determine the frequency and extent of garbage collection to their default settings.

normal-gc is generally used in conjunction with avoid-gc, to cancel the effects of the latter.

Notes normal-gc is useful only in 32-bit LispWorks. In 64-bit implementations it does nothing and simply returns nil.

See also avoid-gc
get-gc-parameters
gc-if-needed
mark-and-sweep
set-gc-parameters
without-interrupts
with-heavy-allocation
“Memory Management in 32-bit LispWorks” on page 135

*packages-for-warn-on-redefinition* Variable

Summary A list specifying packages whose symbols should be checked on attempted definitions.

Package hcl

Initial value (:implementation)
The variable `%packages-for-warn-on-redefinition` is a list of package names or the keyword :implementation, specifying packages which are "protected". For "protected" packages, LispWorks checks before defining (using any of the definer macros like `cl:defun, cl:defclass` and so on) any external symbol of these packages, and takes the action specified by `%handle-warn-on-redefinition` (which defaults to signaling an error).

The symbol :implementation in `%packages-for-warn-on-redefinition` indicates all of the packages which are part of the LispWorks implementation. That includes all the documented packages, including COMMON-LISP and KEYWORD but excluding some "user" packages like CL-USER and KW-USER, and some packages that are used internally.

For symbol value, setting and rebinding is not checked, but defining using definer macros like `cl:defvar` and `cl:defparameter` is checked.

**Notes**

1. The checking is useful because it is relatively easy to redefine an external symbol by mistake, and it leads to undefined behavior which is difficult to debug. It is therefore a bad idea to remove :implementation from the list. In situations when this is required, you should do it by rebinding `%packages-for-warn-on-redefinition` rather than setting it.

2. You can protect your packages by adding their package names to this list.

3. The check is applied for any definition, whether it is actually a redefinition or not. For example, trying to define the symbol `cl:stream` as a function gives an error (by default), even though `cl:stream` has only a class definition, and trying to define `cl:car` as a class also errors even though `cl:car` has only a function definition.
4. You can check whether a package is an implementation package by using `package-flagged-p` with the keyword `:implementation`.

**Compatibility note**

`:implementation` was new in LispWorks 7.0.

In LispWorks 6.1 and earlier versions, the list could contain only package names, and the initial value was a long list of package names.

**See also**

*`handle-warn-on-redefinition`*

`package-flagged-p`

“Protecting packages” on page 88

---

**parse-float**

**Function**

**Summary**

Parses a float from a string and returns it as float.

**Package**

`hcl`

**Signature**

`parse-float string &key start end default-format => float`

**Arguments**

- `string`: A string
- `start, end`: Bounding index designators for `string`
- `default-format`: One of the atomic type specifiers `short-float`, `single-float`, `double-float`, or `long-float`.

**Values**

- `float`: A float

**Description**

The function `parse-float` parses a float from the substring of `string` delimited by `start` and `end` and returns it as `float`.

If the substring represents an integer or the exponent marker is E or is omitted, then `float` will be of type `default-format`, which defaults to the value of `*read-default-float-format*`.

---

975
The HCL Package

...mat*. Otherwise, its type will match the exponent marker as specified by 2.3.2.2 "Syntax of a Float" in the Common Lisp standard.

If the substring does not represent an integer or a float, then an error of type parse-error is signaled.

Examples

(parse-float "10") => 10.0f0
(parse-float "10" :default-format 'double-float) => 10.0d0
(parse-float "10d0") => 10.0d0
(parse-float "10.5") => 10.5f0
(parse-float "10.5d0") => 10.5d0

position-in-ring position-in-ring-forward

Functions

Summary
Finds the first ring element that matches a supplied item and returns its index.

Package
hcl

Signature
position-in-ring ring item index &key test => result
position-in-ring-forward ring item index &key test => result

Arguments
ring A ring object created by make-ring.
index A non-negative integer.
item A Lisp object.
test A function designator for a function of two arguments.

Values
result A non-negative integer or nil.
Description

The function \texttt{position-in-ring} finds in the ring \texttt{ring} the first element that matches \texttt{item} and returns its index, or \texttt{nil} if there is no match. The search starts from index \texttt{index} and proceeds "backward" up to the length of the ring (its current number of elements). In other words, it tests all the elements that would be returned by \texttt{ring-ref} with indices \texttt{index}, \texttt{index+1}, ..., \texttt{length-1}. It does not wrap around, so elements between indices 0 and \texttt{index} are not tested.

The function \texttt{position-in-ring-forward} does the same except that it searches from \texttt{index} "forward" to the insertion point. In other words, it tests the elements that would be returned by \texttt{ring-ref} with indices \texttt{index}, \texttt{index-1}, ..., 0.

The comparison is done by calling \texttt{test}, with \texttt{item} as first argument and each element in the ring as the second argument. The default value of \texttt{test} is \texttt{eql}.

Notes

\texttt{test} is called with the ring locked.

Compatibility notes

In LispWorks 6.1 and earlier versions, these functions are called \texttt{find-in-ring} and \texttt{find-in-ring-forward}. They have been renamed to match the Common Lisp convention that a function returning an index is named \texttt{position-*}. The old names are retained for backwards compatibility, but are deprecated.

See also

\texttt{make-ring}  
\texttt{map-ring}

\textbf{print-profile-list}

Function

Summary

Prints a report of symbols that have been profiled.

Package

\texttt{hcl}

Signature

\texttt{print-profile-list \&key sort limit cutoff collapse => nil}
The HCL Package

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sort</code></td>
<td>:call, :profile or :top</td>
</tr>
<tr>
<td><code>limit</code></td>
<td>An integer.</td>
</tr>
<tr>
<td><code>collapse</code></td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td><code>cutoff</code></td>
<td>A real number.</td>
</tr>
</tbody>
</table>

Values

`print-profile-list` returns `nil`.

Description

The function `print-profile-list` prints a report of symbols, after profiling using `profile`, or `start-profiling` followed by `stop-profiling`.

If the profiler was set up with `style :tree`, then a tree of calls is printed first, according to `limit`, `cutoff` and `collapse`. Then a columnar report is printed showing how often each function was called, profiled and found on the top of the stack. This report is sorted by the column indicated by the value of `sort`.

If the profiler was set up with `style :list`, then only the columnar report is printed.

`sort` can take these values:

- `:call` Sort by the number of times the function was called.
- `:profile` Sort by the number of times the function was found on the stack.
- `:top` Sort by the number of times the function was found at the top of the stack.

If `sort` is not passed then the results are printed as after the profiling run. The default is the value of the variable `*:default-profiler-sort*`.

`limit` is the maximum number of lines printed in the columnar report as described for `*:default-profiler-limit*`. The default is the value of the variable `*:default-profiler-limit*`. 
**cutoff** is the minimum percentage that the profiler will display in the output tree as described for **default-profiler-cutoff**. The default is the value of the variable **default-profiler-cutoff**.

**collapse** controls collapsing of the output tree as described for **default-profiler-collapse**. The default is the value of the variable **default-profiler-collapse**.

**Notes**

You should not call **print-profile-list** while the profiler is running (see **profile** and **start-profiling**) or suspended (see **stop-profiling**).

**Example**

First set up the profiler:

```
CL-USER 1 > (set-up-profiler
    :symbols
    '(cadr car eql fixnump + 1+ caadr cddr))

CL-USER 2 > (profile (dotimes (a 1000000 nil)
    (+ a a)
    (car '(foo))))
```

Then call **print-profile-list**:
CL-USER 3 > (print-profile-list :sort :call)

profile-stacks called 327 times

Cumulative profile summary
Symbol (%) top (%)
called profile

CADR 5000012 13
( 4) 13 ( 4)
CDDR 3000000 3
( 1) 3 ( 1)
EQL 2000202 4
( 1) 4 ( 1)
FIXNUMP 2000003 2
( 1) 2 ( 1)
CAR 1000000 1
( 0) 1 ( 0)
+ 1000000 3
( 1) 3 ( 1)
CAADR 1000000 2
( 1) 2 ( 1)
1+ 1000000 2
( 1) 2 ( 1)

Top of stack not monitored 91% of the time
NIL

Notes
You can suppress printing of those symbols that are currently profiled but which were not called in the profiling run by setting system:*profiler-print-out-all* to nil.

system:*profiler-print-out-all* is a variable defined when the profiler is loaded by set-up-profiler. Its initial value is nil.

See also
*default-profiler-collapse*
*default-profiler-cutoff*
*default-profiler-limit*
*default-profiler-sort*
**print-string**  

**Variable**

Summary: Specifies a maximum length when printing strings.

Package: hcl

Initial value: t

Description: The variable `*print-string*` controls whether the printer uses an abbreviated form for strings when `*print-escape*` is true.

If the value of `*print-string*` is `t` then strings are printed in full as specified by ANSI Common Lisp.

If the value of `*print-string*` is `nil`, then strings are printed as unreadable objects with no specific information about the string.

If the value of `*print-string*` is an integer, then strings longer than `*print-string*` are printed as unreadable objects that include the type, length and first `*print-string*` characters.

**profile**  

**Macro**

Summary: Profile while executing some forms.

Package: hcl

Signature: `profile &body forms => results-of-final-form`

Arguments: `forms` Lisp forms.

Values: `results-of-final-form` Results of the final form in `forms`.
### Description

The macro `profile` starts the profiler, evaluates `forms`, stops the profiler, reports the results of the profiling, and then returns the results of the last form in `forms`.

The profiler is described in Chapter 12, “The Profiler”.

### Note

`profile` profiles all processes. For better control of this, use `start-profiling` or `do-profiling`.

If `profile` is invoked before `set-up-profiler` was ever called, it calls `set-up-profiler` implicitly without arguments.

`profile` cannot be called while another profiling operation is running, either by a parallel call to `profile` or `start-profiling`.

### See also

- `start-profiling`
- `stop-profiling`
- `set-up-profiler`
- Chapter 12, “The Profiler”
- “Guidance for control of the memory management system” on page 130

### *profiler-threshold*

#### Variable

**Summary**

Controls which symbols are profiled on repeated profiling runs.

**Package**

`hcl`

**Description**

`*profiler-threshold*` is used with repeated profiling runs, to control which symbols are profiled. It is set by `set-profiler-threshold`.

**See also**

`set-profiler-threshold`
*profile-symbol-list*  

Variable

Summary  Deprecated. The list of symbols to be profiled.

Package  hcl

Description  *profile-symbol-list* is the list of symbols that are profiled if *profile* is called. Symbols in this list are monitored by the profiler to see if their function objects are on the stack when the profiler interrupts the Lisp process. The length of this list does not affect the speed of the profiling run.

Initial value  nil

Notes  *profile-symbol-list* should normally be set by one of the above functions which check that the symbol is suitable for profiling before adding them to the list.

See also  
add-symbol-profiler  
remove-symbol-profiler  
set-up-profiler

profiler-tree-from-function  

Function

Summary  Prints a call tree of profiled code below a given function.

Package  hcl

Signature  

Arguments  
function-name  A symbol naming a function.  
max-depth  A number or *nil.*
The function `profiler-tree-from-function` prints a tree with root `function-name` whose children are the callees of `function-name` and their callees.

`profiler-tree-from-function` uses the data from the previous 'profile session' with style `:tree`. A profile session ends at the end of `profile` or when `stop-profiling` is called, or when the Profiler tool finishes profiling.

In both cases the counts of profile calls is the total counts of the calls to `function-name`. Note that the percentages (the number in parentheses) are percentages from the total number of profile calls, rather than from the numbers of calls to `function-name`.

If `max-depth` is a number it limits the depth of tree that is printed to that value. The default value of `max-depth` is `nil`, meaning no limit on the depth that is printed.

See also

- profile
- start-profiling
- stop-profiling
  “Profiler output” on page 170

---

**profiler-tree-to-function**

**Summary**
Prints a reversed call tree of profiled code below a given function.

**Package**
hcl

**Signature**

```
profiler-tree-to-function function-name &optional max-depth
```

**Arguments**

- `function-name` A symbol naming a function.
- `max-depth` A number or `nil`. 
### Description

The function `profiler-tree-to-function` prints a tree with root `function-name` whose children are the callers of `function-name` and their callers. Note that the tree is reversed, that is, callers appear under their callees.

`profiler-tree-to-function` uses the data from the previous 'profile session' with style :tree. A profile session ends at the end of `profile` or when `stop-profiling` is called, or when the Profiler tool finishes profiling.

In both cases the counts of profile calls is the total counts of the calls to `function-name`. Note that the percentages (the number in parentheses) are percentages from the total number of profile calls, rather than from the numbers of calls to `function-name`.

`max-depth` limits the depth of tree that is printed. If `max-depth` is `nil` there is no limit on the depth that is printed. The default value of `max-depth` is 7.

### See also

- `profile`
- `profiler-tree-from-function`
- `stop-profiling`
- “Profiler output” on page 170

---

**reduce-memory**

**Function**

**Summary**

Attempts to reduce the size of the Lisp image, without enlarging it even temporarily.

**Package**

hcl

**Signature**

`reduce-memory &optional full => new-size`

**Arguments**

- `full` : `nil` or `t` (or 0, 1, 2 or :aggressive on Mobile GC).

**Values**

- `new-size` : A positive integer.
The function `reduce-memory` frees memory and tries to reduce the size of the Lisp image, without enlarging it even temporarily.

`reduce-memory` has the same effect as `clean-down`, except that `clean-down` may temporarily increase the size of the image in order to be able to promote from lower generations. `reduce-memory` never increases the image size, which means that it may fail to promote. This will cause future garbage collections to be slower, until the promotion actually occurs.

`reduce-memory` is intended to be used when the operating system signals that the memory is low, which is a common feature of mobile platforms, for example `onTrimMemory` and `onLowMemory` in Android and `didRceiveMemoryWarning` in iOS. Using `clean-down` in this situation may cause a temporary increase in size, which may cause the system to run out of memory, or maybe just kill the Lisp process. In other circumstances `clean-down` should do a better job (and you might also consider `try-move-in-generation`).

In 32-bit LispWorks, if `full` is `nil`, `reduce-memory` frees memory and promotes live objects to generation 2. When `full` is non-nil, `reduce-memory` frees and promotes to generation 3.

In ordinary (Sparse) 64-bit LispWorks, `full` is ignored. The call just frees what it can free easily.

When using the Mobile GC, if `full` is `nil`, `reduce-memory` just frees what it can free easily. If `full` is `t`, `reduce-memory` performs a garbage collection on generation 2 and then frees what it can free easily. If `full` is `:aggressive`, `reduce-memory` performs one or more garbage collections until memory is no longer being freed and then frees what it can free easily. When `full` is an integer (0, 1 or 2), it specifies a generation number to garbage collect and `reduce-memory` garbage collects this generation and then frees what it can free easily. Using 2 is the same as using `t`.

The default value of `full` is `nil`. 
reduce-memory returns the new size of the Lisp image after reduction, in bytes.

Notes

1. The default of full is nil, which is different from clean-down where it defaults to t.

2. In 32-bit LispWorks, reduce-memory with no argument or nil differs from (clean-down nil) by trying to reduce the memory. (clean-down nil) frees and promotes, but does not try to reduce the size (and may actually increase it).

3. When using the Mobile GC, reduce-memory releases any reserved memory that the system keeps. As a result any following reduce-memory with argument non-nil will be less effective because there will be no reserved memory to perform copying garbage collection.

See also clean-down
try-move-in-generation (32-bit only)

references-who Function

Summary
Lists special variables referenced by a definition.

Package hcl

Signature references-who function => result

Arguments function A symbol or a function dspec.

Values result A list.

Description The function references-who returns a list of the special variables referenced by the definition named by function.
Notes  The cross-referencing information used by references-who is generated when code is compiled with source-level debugging switched on.

See also  binds-who
toggle-source-debugging
sets-who
who-references

**remove-special-free-action**  

*Function*  

**Summary**  
Removes the specified function from the special actions performed when flagged objects are garbage collected.

**Package**  
hcl

**Signature**  
remove-special-free-action function => function-list

**Arguments**  
function  The function to be removed.

**Values**  
function-list  A list of the functions currently called to perform special actions, not including the one just removed.

**Description**  
Removes the specified function from the special actions performed when flagged objects are garbage-collected. (The special actions are added by add-special-free-action.)

**See also**  
add-special-free-action
flag-special-free-action
flag-not-special-free-action
### remove-symbol-profiler

**Function**

**Summary**
Deprecated. Removes a symbol from the list of profiled symbols.

**Package**
hcl

**Signature**
`remove-symbol-profiler symbol => nil`

**Arguments**
symbol
A symbol to be removed from the *profile-symbol-list*.

**Values**
Returns nil.

**Description**
The function `remove-symbol-profiler` is deprecated. It removes a symbol from the list of profiled symbols.

**See also**
`add-symbol-profiler`

### reset-profiler

**Function**

**Summary**
Resets the profiler so that symbols below a given threshold are no longer profiled.

**Package**
hcl

**Signature**
`reset-profiler &key according-to => nil`

**Arguments**
according-to
One of two values — :profile or :top. This refers to which column of the profiling results `reset-profiler` uses to determine which symbols to delete from the list of symbols to profile. The default is :profile.

**Values**
`reset-profiler` returns nil.
Description  This function updates the list of symbols being profiled according to the results of the previous profiling run. `reset-profiler` runs down the list of symbols being profiled and removes any symbols whose appearance in the previous profiling run falls below the value `*profiler-threshold*`. In this way the number of symbols being considered by the profiler can be reduced to just those which are important.

Example  

```
(reset-profiler :according-to :top)
```

Notes  Reducing the number of symbols in the list of symbols to profile does not actually speed up the execution of the form being profiled, but does reduce the setting up time of the profiler and the size of the list of results.

See also  

- `profile`
- `*profiler-threshold*`
- `print-profile-list`
- `set-profiler-threshold`

---

**reset-ring**

**Function**

**Summary**  Resets a ring.

**Package**  hcl

**Signature**  

```
reset-ring ring => nil
```

**Arguments**  

- `ring`  

  A ring object created by `make-ring`.

**Description**  The function `reset-ring` resets the ring, that is it makes `ring` completely empty.

**See also**  

- `make-ring`
### ringp

**Function**

The predicate for rings.

**Summary**

The predicate for rings.

**Package**

hcl

**Signature**

```
ringp object => result
```

**Arguments**

- `object` A Lisp object.

**Values**

- `result` A boolean.

**Description**

The function `ringp` is a predicate for rings, that is objects created by `make-ring`.

**See also**

`make-ring`

### ring-length

**Function**

Gets the element count and maximum size of a ring.

**Summary**

Gets the element count and maximum size of a ring.

**Package**

hcl

**Signature**

```
ring-length ring => number-of-elements, size
```

**Arguments**

- `ring` A ring object created by `make-ring`.

**Values**

- `number-of-elements` A non-negative fixnum.
- `size` A positive fixnum.

**Description**

The function `ring-length` returns as multiple values the number of elements in the ring ring and its maximum size.

**See also**

`make-ring`
### ring-name

**Function**

**Summary** Returns the name of a ring.

**Package** hcl

**Signature** `ring-name ring => name`

**Arguments**
- `ring` A ring object created by `make-ring`.

**Values**
- `name` A string.

**Description** The function `ring-name` returns the name of the ring `ring`.

See also `make-ring`

### ring-pop

**Function**

**Summary** Removes an element from a ring and returns the element before the insertion point.

**Package** hcl

**Signature** `ring-pop ring &optional remove => object`

**Arguments**
- `ring` A ring object created by `make-ring`.
- `remove` A generalized boolean.

**Values**
- `object` A Lisp object.

**Description** The function `ring-pop` removes (by default) an element from the ring and returns the element before the insertion point.

If `remove` is true then the element is removed from the ring. If `remove` is `nil` then the element remains and instead the ring
is rotated by 1 as if by \( \text{rotate-ring ring 1} \). The default value of remove is \( t \).

\text{ring-pop} signals an error when called on an empty ring.

**Examples**

These 3 forms all return the same values, but the first form removes an element from the ring, while the other two leave all the elements in the ring:

\[
\begin{align*}
\text{(values (ring-pop ring) (ring-ref ring 0))} \\
\text{(values (ring-pop ring t) (ring-ref ring 0))} \\
\text{(values (ring-ref ring 0) (rotate-ring ring 1))}
\end{align*}
\]

**See also**

\text{ring-push}  \\
\text{make-ring}  \\
\text{rotate-ring}  \\
\text{ring-ref}  \\
\text{ring-length}

---

**ring-push**

*Function*

**Summary**

Adds a Lisp object to a ring.

**Package**

\text{hcl}

**Signature**

\text{ring-push object ring => object}

**Arguments**

- \text{object} \hspace{1em} \text{A Lisp object.}
- \text{ring} \hspace{1em} \text{A ring object created by \text{make-ring}.}

**Values**

- \text{object} \hspace{1em} \text{A Lisp object.}

**Description**

The function \text{ring-push} adds any Lisp object as an element of the ring before the “insertion position”, which means that a following \text{ring-pop} will return it. If the ring is full, that is the number of elements in the ring is the same as its size (see
make-ring), then ring-push first removes the element after the insertion point.

Once it finished modifying the ring, if ring-push removed an element and there is a delete-function (see make-ring), then ring-push calls delete-function on the element that it removes.

ring-push returns object.

See also

ring-pop
make-ring
rotate-ring
ring-ref

ring-ref

Function

Summary

Gets or sets the element at a specified offset from the insertion point in a ring.

Package

hcl

Signature

ring-ref ring index => object
(setf ring-ref) object ring index

Arguments

ring A ring object created by make-ring.
index A non-negative integer.
object A Lisp object.

Values

object A Lisp object.

Description

The function ring-ref returns or sets the element at index places before the insertion point in the ring ring.

index must be a non-negative integer smaller than the number of elements in the ring, otherwise an error is signaled.

index 0 returns or sets the element object just before the inser-
tion point, and a larger index goes "back" (in the same direction as ring-pop and rotate-ring).

The setf function replaces the element in the ring with the new element object without affecting the ring otherwise (in particular it does not call delete-function).

See also  
make-ring  
ring-pop  
rotate-ring

rotate-ring  

Function

Summary  Rotates a ring, that is moves the insertion point.

Package  hcl

Signature  rotate-ring ring how-many => object

Arguments  

ring A ring object created by make-ring.

how-many A fixnum.

Values  

object A Lisp object.

Description  The function rotate-ring rotates the ring, that is it moves the insertion point "back", which is the same direction that ring-pop would progress.

how-many is the number of positions to rotate. It has to be a fixnum, but otherwise is not limited.

rotate-ring returns the element before the insertion point after the rotation (the one that (ring-ref ring 0) would return if called immediately after rotate-ring).

Examples  If the ring contains 3 elements or more, then
(progn
  (ring-pop ring)
  (ring-pop ring)
  (ring-ref ring 0))

returns the same value as:

(rotate-ring ring 2)

but the second form does not remove an element from the
ring, while the first form removes 2 elements.

See also
  ring-push
  make-ring
  ring-pop
  ring-ref

safe-format-to-string
safe-format-to-limited-string
safe-prin1-to-string
safe-princ-to-string

Functions

Summary
  Print or format "safely", which means catching errors.

Package
  hcl

Signature
  safe-format-to-string &rest format-args => string
  safe-format-to-limited-string limit &rest format-args => string
  safe-prin1-to-string object => string
  safe-princ-to-string object => string

Arguments
  format-args      A control-string and arguments as passed to format.
  limit            A positive integer.
  object           Any object.
Values  

| string | A string. |

Description  

*safe-format-to-string*, *safe-prinl-to-string* and *safe-princ-to-string* are analogs to the standard functions *format* (with first argument *nil*), *prinl-to-string* and *princ-to-string*. If *format-args* and *object* can be printed without any errors then they are equivalent to the standard functions, except that they bind *print-readably* and *print-circle* to *nil*.

The difference is when there is an error during the printing operation. The "safe" functions catch the error, and try to produce something that indicates that an error occurred during the printing operation and what it was, without causing recursive errors.

*safe-format-to-limited-string* is like *safe-format-to-string*, except that the length of the result *string* is limited to *limit*. The printing is stopped when the output become longer than *limit* and the result is a string of length *limit*, with the last three characters being "...". Limiting the length in this way also copes well when printing deeply nested objects.

Notes  

These functions are intended to be used in code that handles and reports errors, where it is important to avoid recursive errors.

The debugging tools of the LispWorks IDE use these functions.

See also  

*prinl-to-string*  
*princ-to-string*  
*format*

**save-argument-real-p**

Function

Summary  

Deprecated. Returns *t*. 
Package  hcl

Signature  save-argument-real-p => realp

Arguments  None

Values  realp          A boolean.

Description  The function save-argument-real-p always returns t.

Compatibility note  In LispWorks 6.1 for Macintosh and earlier versions, save-argument-real-p can be used to determine whether a build script knows the real name of the image being saved. The return value realp is nil only when building an intermediate image for the purpose of building a universal binary.

In LispWorks 7.0 and later versions, universal binaries are not supported hence save-argument-real-p always returns t.

See also  save-universal-from-script
          building-universal-intermediate-p
          deliver
          save-image
          save-image-with-bundle

save-current-profiler-tree  

Function

Summary  Save the current profiler tree to a file.

Package  hcl

Signature  save-current-profiler-tree filename name => path

Arguments  filename            A pathname designator.
            name                An object.
Values  

<table>
<thead>
<tr>
<th></th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A pathname.</td>
</tr>
</tbody>
</table>

Description  
The function `save-current-profiler-tree` checks if `filename` has a type, and if not adds the type "tree". It then opens the file for writing with `:if-exists :supersede` and `:external-format :utf-8`, and writes the current profiler tree into it. `name` is written to the file as the name of the tree, using `format` with `-A`.

The current profiler tree is set either when `profile` finishes successfully, or when `stop-profiling` is called with `nil` for its `suspend` argument (the default). There is only one current tree, and it is overwritten each time it is set.

The intention of the file is that you can load it into the Profiler tool in the LispWorks IDE to view its contents. However, you can also parse it yourself, or view it as a text file for simple queries.

`name` should be useful for you to remember what the tree is about. For example, it may be something like the result of:

```
(string-append "Computing this and that at "
               (date-string))
```

The Profiler tool displays `name` in the message area of the interface (at the bottom), and it is used in the `History` menu.

`save-current-profiler-tree` returns the path that was used.

The format of the file is described in “Profiler tree file format” on page 174.

See also  

- `stop-profiling`
- `profile`
- “Profiler tree file format” on page 174
**save-current-session**  
*Function*

**Summary**  
Saves the LispWorks session.

**Package**  
hcl

**Signature**  
```
save-current-session pathname &rest save-image-args => result
```

**Arguments**  
- **pathname**  
  A pathname designator.
- **save-image-args**  
  Arguments.

**Values**  
- **result**  
  A boolean.

**Description**  
The function `save-current-session` closes all windows and stops multiprocessing, saves an image at the location supplied in `pathname`, and restarts multiprocessing and the windows. For more information see “Saved sessions” on page 182.

`save-image-args` are passed to the saving function, which is `save-image` on Windows, GTK and Motif, or `save-image-with-bundle` on Cocoa.

`save-current-session` returns `nil` if the pathname supplied is unacceptable (not writable), otherwise it returns `t`. The actual operation is done asynchronously.

**Notes**

1. `save-current-session` is intended for saving the state of a windowing image. While `save-current-session` can be used to save a session in a console image, this achieves nothing more than `save-image`.

2. The released LispWorks image runs the default session. Therefore after you have used `save-current-session`, starting the supplied image (for example via the Windows start menu or MacOS X Dock) will run itself only if the default session is "LispWorks Release".
See also

\texttt{save-image}
\texttt{save-image-with-bundle}

\textbf{save-image} \textit{Function}

\textbf{Summary}
Saves the image to a new file.

\textbf{Package}
\texttt{hcl}

\textbf{Signature}
\texttt{save-image filename &key dll-exports dll-added-files dll-extra-link-options automatic-init gc type normal-gc restart-function multiprocessing console environment remarks clean-down image-type split => nil}

The \texttt{console} argument is available only in LispWorks for Windows and LispWorks for Macintosh.

\textbf{Arguments}
\begin{itemize}
  \item \texttt{filename} A string. It is the name of the file that the image is saved as. This name should not be the same as the original name of the image.
  \item \texttt{dll-exports} A list of strings, or the keyword \texttt{:default}.
  \item \texttt{dll-added-files} A list of strings.
  \item \texttt{dll-extra-link-options} A list of strings.
  \item \texttt{automatic-init} A generalized boolean.
  \item \texttt{gc} If non-nil, there is a garbage collection before the image is saved. The default value is \texttt{t}.
  \item \texttt{type} Determines if some global variables are cleared before the image is saved. You can generally use the default value, which is \texttt{:user}.
  \item \texttt{normal-gc} If this is \texttt{t} the function \texttt{normal-gc} is called before the image is saved. The default is \texttt{t}.
\end{itemize}
restart-function  A function to be called on restart.

multiprocessing  Controls whether multiprocessing is enabled on restart. Possible values are discussed below.

console  On Windows console controls whether the new image will be a Console or GUI application and when, if ever, to make a console window in the latter case.

On the Macintosh console controls when, if ever, to make a console window.

Possible values are discussed below.

environment  environment controls whether the LispWorks environment is started on restart. Possible values are discussed below.

remarks  remarks adds a comment to the save history. The value should be a string.

clean-down  When t, calls (clean-down t).

image-type  One of the keywords :exe, :dll and :bundle.

split  A generalized boolean. If non-nil, the Lisp heap and the executable are saved in two separate files.

Values  Returns nil.

Description  The function save-image saves the LispWorks image to a new executable or dynamic library containing any modifications you have made to the supplied image.

For information about the sort of changes you might want to save in a new image, see Chapter 13, “Customization of LispWorks”.

Do not use save-image when the graphical IDE is running. Instead create a build script and use it with the -build com-
mand line argument similar to the examples below, or run LispWorks in a subprocess using the Application Builder tool.

You cannot use **save-image** on Windows, Linux, FreeBSD, AIX, x86/x64 Solaris and Mac OS X when multiprocessing is running. It signals an error in this case.

On Cocoa you can combine a call to **save-image** with the creation of an application bundle containing your new LispWorks image, as in the example shown below.

**dll-exports** is implemented only on Windows, Linux, x86/x64 Solaris, Macintosh and FreeBSD. It controls whether the image saved is an executable or a dynamic library (DLL). The default value is **:default** and this value means an executable is saved. The value **:com** is supported on Microsoft Windows only (see below). Otherwise **dll-exports** should be list (potentially **nil**). In this case a dynamic library is saved, and each string in **dll-exports** names a function which becomes an export of the dynamic library and should be defined as a Lisp function using **fli:define-foreign-callable**. Each exported name can be found by **GetProcAddress** (on Windows) or **dlsym** (on other platforms). The exported symbol is actually a stub which ensures that the LispWorks dynamic library has finished initializing, and then enters the Lisp code.

On Microsoft Windows the **dll-exports** list can also contain the keyword **:com**, or **dll-exports** can simply be the keyword **:com**, both of which mean that the DLL is intended to be used as a COM server. See the *LispWorks COM/Automation User Guide and Reference Manual* for details.

On Mac OS X the default behavior is to generate an object of type "Mach-O dynamically linked shared library" with file type **dylib**. See **image-type** below for information about creating another type of library on Mac OS X.
On Linux, Macintosh, x86/x64 Solaris and FreeBSD, to save a dynamic library image the computer needs to have a C compiler installed. This is typically `gcc` (which is available by installing Xcode on the Macintosh).

An image saved as a dynamic library (DLL):

- always runs multiprocessing, and
- may need to be shut down by `QuitLispWorks` or by a callback which uses `dll-quit`.

`automatic-init` specifies whether a LispWorks dynamic library should initialize inside the call to `LoadLibrary` (on Microsoft Windows) or `dlopen` (on other platforms), or wait for further calls. Automatic initialization is useful when the dynamic library does not communicate by function calls. On Microsoft Windows it also allows `LoadLibrary` to succeed or fail according to whether the LispWorks dynamic library initializes successfully or not. Not using automatic initialization allows you to relocate the library if necessary using `InitLispWorks`, and do any other initialization that may be required. The default value of `automatic-init` is `t` on Windows, `nil` on other platforms. For more information about automatic initialization in LispWorks dynamic libraries, see Chapter 14, “LispWorks as a dynamic library”.

`dll-added-files` should be a list of filenames. It is ignored on Microsoft Windows. On other platforms if `dll-added-files` is non-nil then a dynamic library containing each named file is saved. Each file must be of a format that the default C compiler (`scm:*c-default-compiler*`) knows about and can incorporate into a shared library. Typically they will be C source files, but can also be assembler or object files. They must not contain exports that clash with names in the LispWorks dynamic library (see Chapter 54, “Dynamic library C functions” for the predefined exports). The added files are useful to write wrappers around calls into the LispWorks dynamic library. Such wrappers are useful for:
• Calling `InitLispWorks` when required, for example to relocate the LispWorks dynamic library to avoid memory clashes with other software, as described under “Startup relocation” on page 459.

• Calling `QuitLispWorks` when required.

• Changing calls that involve complex C structs or even C++ objects into plain calls, because accessing C structures in Lisp requires defining the structure, while in C it only needs to include the header.

• Creating 'stub' functions that can be called from Lisp, for example for calling a C++ method. The address of the stub function can be passed to Lisp which can call it using a function defined by `fli:define-foreign-funcallable`.

• Adding code that runs automatically inside the call to `dlopen`, by using `__attribute__((constructor))`.

- `dll-extra-link-options` should be a list of strings. It is ignored on Microsoft Windows. On other platforms if `dll-extra-link-options` is non-nil then the strings are passed as extra command line arguments to the linker. See the documentation for the linker (typically called ld) for the meaning of these arguments. On Macintosh, a default value for the `-install_name` option is generated using the file-namestring of the dynamic library if "-install_name" is not specified in `dll-extra-link-options`.

- `image-type` defaults to :exe or :dll according to the value of `dll-exports` and therefore you do not normally need to supply `image-type`.

- `image-type :bundle` is used only when saving a dynamic library. On Mac OS X it generates an object of type "Mach-O bundle" and is used for creating shared libraries that will be used by applications that cannot load dylibs (FileMaker for example). It also does not force the filename extension to be `dylib`. On other Unix-like systems `image-type` merely has the
effect of not forcing the file type of the saved image, and the format of the saved image is the same as the default. On Microsoft Windows image-type :bundle is ignored.

**Note:** image-type :bundle is completely unrelated to the Mac OS X notion of an application bundle.

If `split` is `nil` (the default), then the saved image is written as a single executable file containing the Lisp heap. If `split` is `t`, then the saved Lisp heap is split into a separate file, named by adding `.lwheap` to the name of the executable. When the executable runs, it reloads the Lisp heap from this file automatically.

In addition, when saving LispWorks on the Macintosh as an application bundle (for example by using `create-macos-application-bundle`) or as a framework bundle, `split` can be the symbol :resources. This places the Lisp heap file in the Resources directory of the bundle, which allows the heap to be included in the bundle’s signature. For an application bundle, the Resources directory is in the Contents directory alongside the MacOS directory. For a framework bundle, the Resources directory is alongside the shared library. The executable and Lisp heap file must be in these directories within the bundle at run time.

The main use of `split` is to allow third-party code signing to be applied to the executable, which is often not possible when saving an image with the Lisp heap included in a single file.

`restart-function`, if non-nil, specifies a function (with no arguments) to be called when the image is started. If multiprocessing is true, `restart-function` is called in a new process. `restart-function` is called after the initialization file is loaded. The default value of `restart-function` is `nil`.

**Note:** `restart-function` is not called if the command line argument -no-restart-function is present

When multiprocessing is `nil`, the executable image will start without multiprocessing enabled. When multiprocessing is
true or the image is a DLL, the image will start with multiprocessing enabled, starting processes in the list *initial-processes*. When *initial-processes* is nil or multiprocessing is :with-tty-listener, a TTY listener process is started as well. The default value of multiprocessing is nil.

`console` is implemented only in LispWorks for Windows and LispWorks for Macintosh. The possible values for `console` are as follows:

:default
Unchanged since previous save.

t
On the Macintosh, the value t has the same effect as the value :always.
On Windows, a Console application is saved, else a Windows application is saved which creates its own console according to the other possible values.

:input, :output, :io
Whenever input, output or any I/O is attempted on *terminal-io*.

:init
At startup, if input and output are not redirected

:always
At startup, even if input and output are redirected.

The LispWorks for Windows and LispWorks for Macintosh images shipped have `console` set to :input.

The possible values for `environment` are as follows:

:default
Unchanged since previous save.

nil
Start with just the TTY listener.

:t
Start the environment automatically, no TTY listener.
:with-tty-listener

Start the environment automatically, but still have a TTY listener.

The LispWorks image shipped is saved with :environment t on all platforms except for the Motif images on Mac OS X, and SPARC Solaris.

You should not try to save a new image over an existing one. Always save images using a unique image name, and then, if necessary, replace the new image with the old one after the call to save-image has returned.

Notes

1. Do not supply :multiprocessing nil along with a true value of :environment t. Multiprocessing is needed for the GUI environment.

2. load-image checks that load-all-patches has been called. If it is called before load-all-patches in a session then an error is signaled.

3. In the example build scripts below, the call to load-all-patches is not strictly required when the script is used with the -build command line argument, because LispWorks 6.1 and later versions call load-all-patches automatically. However, it does no harm for the build script to call load-all-patches too.

Compatibility notes

1. LispWorks 5.0 and previous versions documented -init as the way to run LispWorks with a build script. This way is deprecated.

2. Note that LispWorks quits automatically after processing a build script via -build, whereas with -init you need to call quit explicitly at the end of the build script.

3. In LispWorks 5.0 and previous versions dll-exports is supported only on Windows.

4. dll-added-files and automatic-init were new in LispWorks 5.1.
Example

Here is an example build script. Save this to a file such as 
\texttt{c:/build-my-image.lisp}:

\begin{verbatim}
(load-all-patches)
(load "my-code")
(save-image "my-image")
\end{verbatim}

Then run LispWorks with the command line argument 
\texttt{-build c:/build-my-image.lisp} to save the image 
\texttt{my-image.exe}.

This example shows a portable build script which, on Cocoa, 
saves your new LispWorks image in a Mac OS X application 
bundle. This allows your new LispWorks for Macintosh 
image to be launchable from the Finder or Dock and to have 
its own icon or other resources:

\begin{verbatim}
(load-all-patches)
(load "my-code")
#:cocoa
(compile-file-if-needed
  (example-file
    "configuration/macos-application-bundle")
  :load t)
(save-image
 #:cocoa
  (write-macos-application-bundle
   "Applications/LispWorks 7.1/My LispWorks.app")
=:cocoa
 "my-lispworks")
\end{verbatim}

See also

\begin{verbatim}
deliver
dll-quit
*initial-processes*
InitLispWorks
LispWorksDlsym
load-all-patches
quit
QuitLispWorks
save-current-session
"Guidance for control of the memory management system"
on page 130
\end{verbatim}
**save-image-with-bundle**

*Function*

**Summary**
Saves a LispWorks for Macintosh image with an application bundle, thus allowing it to work properly in the Cocoa windowing system.

**Package**
hcl

**Signature**
```
save-image-with-bundle bundle-path &rest save-image-args &key bundle-arguments bundle-function &allow-other-keys
```

**Arguments**
- **bundle-path** A pathname designator.
- **save-image-args** Arguments passed to `save-image`.
- **bundle-arguments** Arguments passed to `bundle-function`.
- **bundle-function** A function designator.

**Description**
The function `save-image-with-bundle` first creates the application bundle using the function `bundle-function`, and then saves the LispWorks image in the bundle.

The default value of `bundle-arguments` is `nil`.

The default value of `bundle-function` is `create-macos-application-bundle`. You can modify the created bundle by supplying `bundle-arguments`.

With the default values of `bundle-function` and `bundle-arguments`, it copies the application bundle of the running image to the bundle path with the minimal necessary modifications, and then saves an image in it.

`save-image-with-bundle` operates as follows:

1. It calls `bundle-function` with the `bundle-path` and `bundle-arguments`, and then uses the result as the filename for `save-image`. 
2. It applies `save-image` to the path derived in the first step and the remaining arguments passed to `save-image-with-bundle` (other than `bundle-arguments` and `bundle-function`).

Notes

`save-image-with-bundle` is implemented only in LispWorks for Macintosh.

See also

`create-macos-application-bundle`
`save-image`

### save-universal-from-script

**Function**

**Summary**


**Package**

`hcl`

**Signature**

```
save-universal-from-script target-image script-name &key
  output-stream => target-image
```

**Arguments**

- `target-image` A pathname designator.
- `script-name` A pathname designator.
- `output-stream` A stream or `nil`.

**Values**

`target-image` A pathname designator.

**Description**

The function `save-universal-from-script` is deprecated.

**Compatibility note**

In LispWorks 6.1 for Macintosh and earlier versions, `save-universal-from-script` provides a convenient way to create a universal binary on an Intel Macintosh, using a script designed for saving a mono-architecture image.
LispWorks 7.0 and later versions do not support universal binaries, hence `save-universal-from-script` now simply signals an error.

See also

- `save-image`
- `create-universal-binary`
- `building-universal-intermediate-p`
- `save-argument-real-p`

### set-array-single-thread-p

**Function**

**Summary**

Tells the system whether an array is accessed only in a single thread context, or not.

**Package**

`hcl`

**Signature**

`set-array-single-thread-p array on-p`

**Arguments**

- `array` An array.
- `on-p` A generalized boolean.

**Description**

Tells the system whether the array `array` is accessed only in a single thread context or not, depending on the value of `on-p`. Arrays that are marked for single thread access are faster for some operations, in particular `vector-push` and `vector-pop`.

See also

- `array-single-thread-p`
- `make-array`

### set-array-weak

**Function**

**Summary**

Sets the weakness state of an array.
Package       hcl
Signature     set-array-weak array weakp => weakp
Arguments     array    A non-displaced array, with
               array-element-type t.
               weakp    If weakp is non-nil, the array is made weak.
                        If weakp is nil, the array is made non-weak.
Values        Returns weakp.
Description   By default, arrays are non-weak, and they keep alive all the
               objects that are stored in them. A weak array may remove a
               pointer if the object that it points to is not pointed to from
               somewhere else. When a pointer is removed like this, it is
               replaced in array with nil.
               Pointers are replaced by nil after a garbage collector (GC)
               operation that identifies that they can be replaced. This
               means that if the object that is pointed to has been promoted
               to a higher generation, a garbage collection of the higher gen-
               eration is required to remove the pointer. Note that by default
               the system does not automatically GC the blocking genera-
               tion or higher.
               The weakness state of an array can be changed many times.
               In all implementations, array must not be a displaced array,
               and the array-element-type of array must be t.
               In 64-bit LispWorks, an additional requirement is that array
               must be an adjustable array.
               set-array-weak can be called at any moment.
Notes          An array can be made weak at creation time using the :weak
               argument to make-array.
See also       array-weak-p
               copy-to-weak-simple-vector
```lisp
set-default-generation
```

**Function**

**Summary**
Set the current generation for memory allocation in 32-bit LispWorks.

**Package**
hcl

**Signature**
`set-default-generation num => num`

**Arguments**

| num | The number of the generation from which to do future allocation. |

**Values**
Returns `num`.

**Description**
Set the current generation for memory allocation. By default the system allocates memory from the youngest generation (generation 0).

**Notes**
`set-default-generation` is useful only in 32-bit LispWorks. In 64-bit implementations it does nothing and returns 0.

**Examples**
```lisp
(set-default-generation 1)
;; allocate from an older generation
(set-default-generation 0)
;; return to normal
```

**See also**

- `allocation-in-gen-num`
- `clean-generation-0`
- `collect-generation-2`
- `collect-highest-generation`
set-gc-parameters  

**Function**

**Summary**
Sets the parameters for the garbage collector in 32-bit LispWorks. This function is deprecated.

**Package**
hcl

**Signature**
```lisp
(set-gc-parameters &key maximum-buffer-size minimum-buffer-size big-object promote-min-buffer promote-max-buffer new-generation-size minimum-overflow maximum-overflow minimum-for-sweep minimum-for-promote enlarge-by-segments => <no values>)
```

**Arguments**
Unless stated, arguments are in bytes:

- **maximum-buffer-size**
  Maximum size of the small objects buffer.

- **minimum-buffer-size**
  Minimum size of the small objects buffer.

- **big-object**
  An object that is bigger than this value is “big”. That is, it is not allocated from the small objects buffer, but from the big-chunk area (if it is allocated in generation 0 in the normal way).

- **promote-min-buffer**
  During promotion, a buffer is allocated in the generation being promoted into, and the objects promoted are moved into it. **promote-min-buffer** controls the minimum size of this buffer.
**promote-max-buffer**

Controls the maximum size of the promotion buffer.

**new-generation-size**

Controls the minimum enlargement of generation gen-num, for gen-num > 0. Value 0 means the generation is not expanded. Otherwise, new-generation-size must be a fixnum in the exclusive range (10000, 100000000) and the minimum expansion is then new-generation-size * gen-num words.

new-generation-size has no effect on the enlargement of generation 0.

**maximum-overflow**

Maximum size of the small-objects buffer in the big-chunk area.

**minimum-overflow**

Minimum size of the small-objects buffer in the big-chunk area.

**minimum-for-promote**

Controls the frequency of promotions. Setting minimum-for-promote to a high value causes the system to promote less frequently. This may improve performance for programs that allocate a lot of data for a short term and then delete it.

**minimum-for-sweep**

Controls when a mark and sweep takes place. Setting minimum-for-sweep to a high value causes the system to mark and sweep less often, which means it has to grow. The
CPU time spent in garbage collection is mostly smaller, but the process is bigger and may cause more disk access.

**enlarge-by-segments**

A minimum for how much the image grows each time a segment is enlarged, as a multiple of 64 KB. This parameter is ignored when adding a static segment.

**Values** None.

**Description** This function sets the parameters of the garbage collector, using the keywords described above.

**Notes** *set-gc-parameters* is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations.

**See also** *get-gc-parameters*  
“Memory Management in 32-bit LispWorks” on page 135

---

**set-hash-table-weak**

*Function*

**Summary** Sets the weakness state of a hash-table.

**Package** `hcl`

**Signature**

```lisp
(set-hash-table-weak hash-table weak &optional free-function) => weakness-state
```

**Arguments**

- `hash-table` A hash-table.
- `weak` Sets the weakness state of `hash-table`. Value may be:
  - `:value` or `t` — An entry is kept if there is a pointer to the value from another object.
:key — An entry is kept if there is a pointer to the key from another object.

:both — An entry is kept if there are pointers to both the key and the value.

:one or :either — An entry is kept if there is a pointer to either the key or the value.

nil — Make the hash-table non-weak. All entries are kept.

free-function A designator for a function of two arguments.

Values Returns weak, unless t was passed, when :value is returned.

Description By default, hash-tables are not weak, which means that they keep alive all the keys and the values in the table.

A weak hash-table allows entries to be removed if there are no other pointers to them. The weakness-state tells the system which entries may be removed like this.

Entries that can be removed are removed after a garbage collector (GC) operation which identifies that they can be removed. This means that if the relevant object(s) (the key or the value) have been promoted to a higher generation, a garbage collection (GC) of the higher generation is required to remove them from the table. Note that by default the system does not automatically GC the blocking generation or higher.

The weakness-state of a hash-table can be changed repeatedly, at any time, at any point using any of the weak values listed above. It can also be set by make-hash-table.

free-function can be supplied to specify a free function as described for make-hash-table. It has no effect if weak-kind is nil.
See also
hash-table-weak-kind
make-hash-table
mark-and-sweep
set-array-weak
“Freeing of objects by the GC” on page 165

set-minimum-free-space

Function

Summary
Sets the minimum free space for a segment of the specified
generation in 32-bit LispWorks.

Package
hcl

Signature
set-minimum-free-space gen-num size &optional segment =>
generation-size

Arguments

gen-num The generation to be affected.
size The size (in bytes) to set the segment to.
segment An integer specifying the segment to be
affected. The default value is 0, meaning the
first segment of the generation.

Values
generation-size A list showing information for the genera-
tion just specified in the call.

Description
Sets the minimum free space for a segment of the specified
generation.

By default, affects the first segment — pass segment to affect a
different segment of the generation.

The minimum free space is shown by room.

Notes
set-minimum-free-space is implemented only in 32-bit
LispWorks. It is not relevant to the Memory Management
API in 64-bit implementations.
set-process-profiling

**Function**

**Summary**
Controls the set of processes that are profiled.

**Package**
hcl

**Signature**
set-process-profiling flag processes

**Arguments**
flag :add, :remove or :set.
processes One of :current, :all, :new, a mp:process object, or a list of mp:process objects which may also contain :current or :new.

**Description**
The function set-process-profiling modifies the set of processes for which profiling information is (or will be) collected.

If set-process-profiling is called while profiling (that is after a call to start-profiling and before the next call to stop-profiling with print non-nil) the system immediately starts collecting profile information for the new set of processes.

When start-profiling is called without passing processes, it sets the processes to profile according to the last call to set-process-profiling.

flag determines how the set of processes to profile is modified:
:add The given processes are added to the set.

:remove The given processes are removed from the set.

:set The given processes are used as the set.

processes controls which processes are added to the set, removed from the set or are contained in the set, as follows:

:current Means the current process. When start-profiling is called it interprets :current to mean the current process at the time it is called. If set-process-profiling is called while profiling, :current is interpreted as the current process when set-process-profiling is called.

:all Means all processes, including those which are created after profiling started.

:new All processes created after the call to start-profiling, unless set-process-profiling is called while profiling, in which case it is any process created after this call.

A mp:process object
  Means that process.

A list Means the processes in that list. The list can contain the symbols :current or :new, which are interpreted as described above.

set-process-profiling can be called whether or not the profiler is collecting information. See start-profiling and stop-profiling.

Note: This function only works on platforms in SMP LispWorks; in non-SMP LispWorks, all processes are profiled.

Examples Add process1 to the set:

(set-process-profiling :add process1)
Turn off profiling for the current process:

```
(set-process-profiling :remove :current)
```

Turn off all profiling:

```
(set-process-profiling :remove :all)
```

Set all processes for later profiling:

```
(set-process-profiling :set :all)
```

See also

- `profile`
- `start-profiling`
- `stop-profiling`

Chapter 12, “The Profiler”

**set-profiler-threshold**  
*Function*

**Summary** Sets the percentage threshold for symbols to be profiled in a subsequent run.

**Package** 

```
hcl
```

**Signature**

```
set-profiler-threshold value => value
```

**Arguments**

`value` must be a fixnum between 0 and 100.

**Values**

`set-profiler-threshold` returns `value`.

**Description**

This function sets the value of `*profiler-threshold*` below which symbols are not profiled in a repeated profiling run. After a profiling run, all the symbols being profiled have a percentage value for the amount of time they were on the top of the stack. If `*profiler-threshold*` is set to 40 then by running `reset-profiler` with argument `:top` all symbols which are found on the top of the stack less than forty percent of the time are removed from the list of those symbols considered for profiling.
Example: (set-profiler-threshold 40)

See also: reset-profiler
profile
*profiler-threshold*

**set-promotion-count**

*Function*

**Summary**
Controls when objects can be promoted to the next generation in 32-bit LispWorks. This function is deprecated.

**Package**
hcl

**Signature**

```lisp
(set-promotion-count gen-num count &optional segment => count)
```

**Arguments**

- `gen-num` The generation number affected.
- `count` The number of garbage collections survived by objects in that generation, before promotion. If `count` is `nil`, the function returns the current promotion count setting.
- `segment` An integer specifying which segment of the generation is to be affected. The default is 0, meaning the lowest segment of the generation.

**Values**

Returns `count`.

**Description**
Controls how many garbage collections an object in a segment must survive before promotion to the next generation.

**Notes**

1. `set-promotion-count` is deprecated, because experience has shown that it is not useful.
2. set-promotion-count is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations, wherein you may be able to achieve the effect with set-delay-promotion.

See also
- block-promotion
- clean-generation-0
- collect-generation-2
- collect-highest-generation
- expand-generation-1

**set-system-message-log**

*Function*

**Summary**

Manipulates the system message log.

**Package**

hcl

**Signature**

```
set-system-message-log &key stream collect get callback => result
```

**Arguments**

- **stream**
  
  An output stream designator, or :no-change.

- **collect**
  
  A boolean, or :no-change.

- **get**
  
  t or :keep.

- **callback**
  
  A function designator, or :no-change.

**Values**

- **result**
  
  A list of strings, or nil.

**Description**

The function set-system-message-log manipulates the system message log. This log is used by the system to produce messages that indicate that something is not as expected, but is not an error. For example, putting a bad Break-Gesture in a GTK resource file.
If stream is t or a stream, the system message log stream is set, with t meaning *standard-output*. This stream is used when writing messages.

When collect is true but not :no-change, messages are collected in an internal list, which can be retrieved by using get.

callback can be a designator for a function of one argument, a string. This function is called when a message is generated. The callback must not try to perform GUI operations.

The default value of each of stream, collect and callback is :no-change, which does not change the current setting.

When get is supplied set-system-message-log returns a list of the messages that has been collected. Each message is a single string. If get is t, the internal list is reset to nil. If get is :keep, the internal list is not reset, so the next call with get will get them again.

set-system-message-log returns nil if get is not supplied.
set-system-message-log returns the list of collected messages if get is supplied.

Notes

stream, callback and collect are mutually independent. It is possible to set the system to any combination of these.

The order of operation when a message is generated is first to print, then call the callback, and then collect.

When collecting messages it can accumulate, so it is important to periodically get the message to ensure it does not bloat the memory.

Using collect t when it is already collecting has no effect, in particular it does not affect the list of collected messages.

set-up-profiler

Function

Summary
Declares the parameter values of the profiling function.
Package  

hcl

Signature  

set-up-profiler &key symbols packages kind interval limit cutoff collapse style gc call-counter show-unknown-frames kw-contexts subfunctions

Arguments  

symbols A symbol or a list of symbols.

packages A valid package name, or a list of package names, :none or :all.

kind :profile, :virtual or :real.

interval An integer greater than or equal to 10000.

limit An integer or nil.

cutoff An integer or nil.

collapse A generalized boolean.

style :tree, :list or nil.

gc A generalized boolean.

call-counter A generalized boolean.

show-unknown-frames A generalized boolean.

ekw-contexts t or a list of KnowledgeWorks context names.

subfunctions A boolean.

Values  

The time interval is returned.

Description  

set-up-profiler is used to declare the values of the parameters of the profiling function.

packages specifies that the symbols in these packages should be monitored, that is added to *profile-symbol-list*. If packages is :all, then symbols in all packages are monitored. If packages is :none, then no package is used to find symbols
to monitor. Otherwise, packages should be a list of package specifiers, and the symbols in these packages are monitored.

If symbols is non-nil, it should be a list of function-dspecs to monitor in addition to the symbols that were specified by packages. These are typically symbols, but can be other functions as specified in “Function dspecs” on page 84.

Note: When a symbol that names a generic function should be monitored, LispWorks adds all the methods of the generic function to the profile list.

If both packages and symbols are nil, then set-up-profiler behaves as if packages is :all. Thus if you actually want to monitor no symbols, you need to pass :packages :none. That is useful if you want to monitor only KnowledgeWorks rules (see kw-contexts below).

kind specifies the way that the time between samples is measured on Unix-like platforms:

:profile Process time only.
:virtual Process time and system time for the process.
:real Real time.

The default value of kind is :profile.

Note: kind is ignored on Microsoft Windows platforms.

interval specifies the interval in microseconds between profile samples. The minimum value of interval is 10000, that is 10 ms. The default value of interval is 10000.

limit, when non-nil, sets *default-profiler-limit*. This limits the maximum number of lines printed in the profile output (not including the tree). The default value is 100.

cutoff, when non-nil, sets *default-profiler-cutoff*. This is the default minimum percentage that the profiler will display in the output tree. Functions below this percentage
will not be displayed. The default is `nil`, that is there is no cutoff.

collapse specifies whether functions with only one callee in the profile tree should be collapsed, that is, only the child is printed. When passed, sets `*default-profiler-collapse*`. The default value of `collapse` is `nil`.

style controls the format of output. If `style` is not passed or passed as `nil`, the format does not change. If `style` is passed, it can take these values:

- `:list` The profiler will show the functions seen on the stack.
- `:tree` The profiler will generate a tree of calls seen in the profiler, as well as the output shown by `:list`.

The default value of `style` is `:tree`.

gc specifies whether to profile functions inside the memory management code (more accurately, functions that are called on the GC stack) in addition to any other profiling. The default value of `gc` is `nil`.

call-counter specifies whether to add extra code to count calls. The counting is done dynamically. If `call-counter` is `nil`, call counters are not added, and the call counter of all functions is displayed as 0. The default value of `call-counter` is `nil` in SMP LispWorks and `t` in non-SMP LispWorks. This is because the counting significantly affects the performance of applications using Symmetric Multiprocessing (SMP).

show-unknown-frames controls whether the profile tree shows nodes where the name of the function is unknown. The default value of `show-unknown-frames` is `nil`.

kw-contexts allows you to profile forward chaining rules in KnowledgeWorks (see the KnowledgeWorks and Prolog User Guide). When `kw-contexts` is `t` (the default), all context are pro-
filed. Otherwise it should be a list of context names. The profiler profiles all the forward rules in each context.

`subfunctions` controls whether to profile subfunctions of the functions that are profiled. When it is non-nil, for each function that the profiler is profiling, the profiler checks if it has subfunctions, and if it has then it profiles them too. The default value of `subfunctions` is `nil`.

If `subfunctions` is non-nil then initializing the profiler is somewhat slower, and also, because the names of subfunctions are long, the output is more messy. It is sometimes useful though.

Notes

1. If the profiler is invoked before any call to `set-up-profiler`, it calls `set-up-profiler` implicitly without any arguments. That means it will monitor all symbols in the image, and if KnowledgeWorks is loaded also all forward chaining rules. In most cases this is a useful behavior, so it is not necessary to use `set-up-profiler`.

2. `set-up-profiler` finds all the symbols in the specified packages at the time it is called. Thus symbols that are give function definitions after the call to `set-up-profiler` are not profiled, whether or not they are in packages that were passed to `set-up-profiler`. If you want to ensure that all symbols are profiled, you need to call `set-up-profiler` just before invoking the profiler.

3. Call counting can affect performance significantly on some platforms. To get accurate timing (in scales of a few percentage points), pass `call-counter nil`. However, in most cases the profiler is used to find bottlenecks where the slowdown is hundreds of percentage points and so the effect of call counting is less significant.
4. *call-counter* is effective only on x86 platforms or in 64-bit LispWorks. On non-x86 platforms 32-bit LispWorks does call counting for a given function if the compiler optimize quality `debug` is greater than 0 at compile-time, and *call-counter* has no effect.

5. *limit*, *cutoff* and *collapse* affect only the display of the results, not the collection of profiler data.

Examples

```
(set-up-profiler :packages '(my-package common-lisp))
(set-up-profiler :symbols
  (my-list-all-interesting-functions))
```

See also

*default-profiler-collapse*
*default-profiler-cutoff*
*default-profiler-limit*
profile
start-profiling
stop-profiling
Chapter 12, “The Profiler”

**sets-who**

Function

Summary

Lists special variables set by a definition.

Package

hcl

Signature

`sets-who function => result`

Arguments

`function` A symbol or a function dspec.

Values

`result` A list.

Description

The function *sets-who* returns a list of the special variables set by the definition named by `function`. 
Notes
The cross-referencing information used by sets-who is generated when code is compiled with source-level debugging switched on.

See also
binds-who
who-sets
toggle-source-debugging
references-who

source-debugging-on-p

Function
Summary Tests if source level debugging is on for compiled code.
Package hcl
Signature source-debugging-on-p => bool
Arguments None.
Values bool If t, source level debugging is on.
Description Returns t if source level debugging is on for compiled code; otherwise returns nil.
See also toggle-source-debugging

start-gc-timing
stop-gc-timing
get-gc-timing

Functions
Summary Time Garbage Collector (GC) operations.
Package hcl
Signature

```lisp
start-gc-timing &key initialize
stop-gc-timing
get-gc-timing &key reset
```

Arguments

- `initialize` A generalized boolean.
- `reset` A generalized boolean.

Description

The functions `start-gc-timing`, `stop-gc-timing` and `get-gc-timing` time Garbage Collector (GC) operations.

`start-gc-timing` causes the system to start collecting GC timing. If `initialize` is non-nil, `start-gc-timing` also resets the Garbage Collector times to 0. The default value of `initialize` is `t`.

`stop-gc-timing` stops collecting GC timing, but does not affect the times.

`get-gc-timing` returns the GC timing as a plist of the form

```lisp
(:total total :user user :system system)
```

where `total`, `user` and `system` are real numbers giving the total, user and system times in seconds spent inside the Garbage Collector while GC timing is on since the timing was last reset. When `reset` is non-nil, `get-gc-timing` also switches GC timing off and resets the timing to 0. The default value of `reset` is `nil`.

The GC timing is the same timing that is collected by `extended-time`. Once `start-gc-timing` is called, `extended-time` does not try to collect GC timing and print it until `get-gc-timing` is called with `reset` non-nil.

`get-gc-timing` can be called while GC timing is being collected.

Notes

`stop-gc-timing` and `start-gc-timing` (with `initialize = nil`) can be used to collect GC timing only in specific periods without resetting the times. However the points at which the
Garbage Collector is invoked are not well-defined, so the program may allocate while GC timing is on, and spend time Garbage Collecting after you stopped collecting.

See also extended-time
room
time

**start-profiling**

*Function*

**Summary**

Starts collecting profiling information.

**Package**

hcl

**Signature**

`start-profiling &key initialize processes profile-waiting ignore-in-foreign time`

**Arguments**

- `initialize` A boolean.
- `processes` One of :current, :all, a mp:process or a list of mp:process objects.
- `profile-waiting` A boolean.
- `ignore-in-foreign` A boolean.
- `time` `t`, `nil` or `:extended`.

**Description**

The function `start-profiling` starts collecting profiling information.

If `initialize` is non-nil any profiling information collected so far is discarded. The default value of `initialize` is `t`.

If `processes` is supplied, the set of processes that will be profiled is set as if by calling:

```
(set-process-profiling :set :processes processes)
```

Otherwise, the set of processes remains unchanged, so is controlled by any previous calls to `set-process-profiling`. 
processes only works in SMP LispWorks. In non-SMP LispWorks, all processes are profiled.

profile-waiting is used only in SMP LispWorks. When profile-waiting is true, processes that are marked for profiling are profiled even if they are in a wait state. In non-SMP LispWorks, only processes that are active are profiled.

ignore-in-foreign controls whether to ignore processes that are inside foreign calls. The default value of ignore-in-foreign is nil.

time controls whether to output overall timing information with the profiler output. If time is nil then no timing information is output. If time is t (the default), then output like time is printed. If time is :extended, output like extended-time is printed. The output is done when stop-profiling is called with print and suspend nil, which are the defaults.

start-profiling can be repeatedly called without intervening calls to stop-profiling, for example to change the setting of profile-waiting or the profiled processes.

start-profiling cannot be used while profile is used or while the Profiler tool is profiling (on any thread). Between the call to start-profiling and the next call to stop-profiling with print t (or omitted), profile and the Profiler tool cannot be used.

Various parameters which are set by set-up-profiler control the behavior of the profiler. See the documentation for set-up-profiler.

If start-profiling is called before any call to set-up-profiler, it implicitly calls set-up-profiler without arguments, which will cause it to monitor all fbound symbols in the image.

Examples

The following sequence of calls to start-profiling and stop-profiling can be used to profile only interesting work and print the results:
Start profiling the current process:

(start-profiling :processes :current)
(do-interesting-work)

Temporarily suspend profiling:

(stop-profiling :print nil)
(do-uninteresting-work)

Resume profiling:

(start-profiling :initialize nil)
(do-more-interesting-work)
(stop-profiling)

See also

profile
do-profiling
set-process-profiling
stop-profiling

“Guidance for control of the memory management system”
on page 130
Chapter 12, “The Profiler”

string-trim-whitespace

Function

Summary Trims whitespace characters from the beginning and end of a string.

Package hcl

Signature string-trim-whitespace string => trimmed-string

Arguments string A string.

Values trimmed-string A string.

Description The function string-trim-whitespace returns a substring of string, with all whitespace characters stripped off the
beginning and end. A whitespace character is a character for which \texttt{whitespace-char-p} returns \texttt{t}.

See also \texttt{whitespace-char-p} \\
\texttt{cl:string-trim}

\textbf{stop-profiling} \hfill \textit{Function}

\textbf{Summary} \hfill Stops collecting profiling information.

\textbf{Package} \hfill \texttt{hcl}

\textbf{Signature} \hfill \texttt{stop-profiling &key print stream}

\textbf{Arguments} \hfill  \\
\texttt{print} \quad A generalized boolean. \hfill  \\
\texttt{stream} \quad An output stream. \hfill  \\
\texttt{suspend} \quad A generalized boolean.

\textbf{Description} \hfill The function \texttt{stop-profiling} stops collecting profiling information, and optionally prints the results. \\
If \texttt{suspend} is false, then the next call to \texttt{start-profiling} must pass \texttt{initialize t} or omit the \texttt{initialize} argument. In addition, if \texttt{print} is true, then the collected profiler information is printed. \\
If \texttt{suspend} is true, then the profiler is put into a suspended state where no profiling information is collected, but can be restarted by calling \\
(\texttt{start-profiling :initialize nil}) \\
The default value of \texttt{print} is \texttt{t} and the default value of \texttt{suspend} is (not \texttt{print}). The value of \texttt{print} is ignored if \texttt{suspend} is true. \\
\texttt{stream} specifies the stream for output when \texttt{print} is non-nil. It is ignored when \texttt{print} is \texttt{nil}. The default value of \texttt{stream} is the value of \texttt{*trace-output*}. 

Parameters set by `set-up-profiler` control the format of the output.

See also `do-profiling`  
`profile`  
`set-process-profiling`  
`start-profiling`  
“Guidance for control of the memory management system” on page 130  
Chapter 12, “The Profiler”

**sweep-all-objects**  
*Function*

**Summary**  
Applies a function to all the live objects in the image.

**Package**  
hcl

**Signature**  
`sweep-all-objects function &optional gen-0 => nil`

**Arguments**  
`function`  
A function of one argument, the object.  
`gen-0`  
A generalized boolean, default value `nil`

**Values**  
`sweep-all-objects` returns `nil`.

**Description**  
Applies `function` to all the live objects in the image. Normally it is not useful to sweep objects in generation 0 because they are ephemeral, so by default `sweep-all-objects` does not sweep generation 0. This can be changed by passing a non-nil value as `gen-0`.

`function` should take one argument, the object. It can allocate, but if it allocates heavily the sweeping becomes unreliable. Small amounts of allocation will normally happen only in generation 0, and so will not affect sweeping of other generations.
To call `sweep-all-objects` reliably, do it inside `with-other-threads-disabled`.

**Notes**

In 64-bit LispWorks and in the Mobile GC there is a more specific alternative: function `sweep-gen-num-objects` can be used to call a function on all live objects in a particular generation.

In the Mobile GC, `sweep-all-objects` does not sweep `cons` objects. There is also a more specific alternative: function `mobile-gc-sweep-objects` can be used to call a function on all live objects in particular generations.

See also

- `sweep-gen-num-objects`
- `mobile-gc-sweep-objects`

---

**switch-static-allocation**

**Function**

**Summary**

Controls whether objects are allocated in the static area.

**Package**

`hcl`

**Signature**

`switch-static-allocation flag => previous-flag`

**Arguments**

- `flag`

  If `flag` is non-nil, subsequent objects are allocated in the static area; if `flag` has any other value, objects are allocated conventionally.

**Values**

`switch-static-allocation` returns the previous setting of `flag`.

**Description**

Objects in the static area are garbage-collected, but not moved.

You should avoid using this function.
See also  
- enlarge-static
- in-static-area

*symbol-alloc-gen-num*  
**Variable**

**Summary**  Specifies the generation in which interned symbols and their symbol names are allocated.

**Package**  hcl

**Initial value**  2 in 32-bit LispWorks, 3 in 64-bit LispWorks

See also  
- allocation-in-gen-num
- get-default-generation
- set-default-generation
- “Allocation of interned symbols and packages” on page 164

symbol-dynamically-bound-p  
**Function**

**Summary**  The predicate for whether a symbol is dynamically bound.

**Package**  hcl

**Signature**  

symbol-dynamically-bound-p symbol => result

**Arguments**  
- symbol  A non-nil symbol.

**Values**  
- result  A boolean.

**Description**  The function symbol-dynamically-bound-p is the predicate for whether the symbol symbol is dynamically bound in the current environment.

See also  
Chapter 28, “Miscellaneous Utilities”
The HCL Package

### throw-if-tag-found

**Macro**

**Summary**
Throws to a specified catch tag or returns `nil` if the catch tag is not found.

**Package**
`hcl`

**Signature**
`throw-if-tag-found catch-tag result-form => A non-local exit or nil`

**Arguments**
- `catch-tag` A catch tag.
- `result-form` A Lisp form.

**Description**
The macro `throw-if-tag-found` checks whether it can find the catch tag `catch-tag` by using `find-throw-tag`. If it finds `catch-tag` it throws to `catch-tag` the value(s) of evaluating `result-form`. Otherwise `throw-if-tag-found` returns `nil`, without evaluating `result-form`.

The throwing operation is done by a normal throw. Therefore the only the difference between this and `cl:throw` is when the tag is not found. In this case, `cl:throw` would evaluate the result form and then give an error, but `throw-if-tag-found` simply returns nil.

**See also**
`find-throw-tag`

### toggle-source-debugging

**Function**

**Summary**
Changes compiler settings affecting production of source level debugging information.

**Package**
`hcl`

**Signature**
`toggel-source-debugging &optional on => bool`
Arguments on Flag (t or nil) to control the resulting setting of the variables. The default is t.

Values bool The current state of source level debugging: t if source level debugging is on.

Description toggle-source-debugging sets certain compiler parameters, and also turns leaf case optimizations on (when called with nil) or off (when called with t). For all these parameters, the value nil reduces compilation speed.

toggle-source-debugging is called in the configuration file a-dot-lispworks.lisp, and the initial state of LispWorks such that source level debugging is on.

The parameters relate to information required for source level debugging, cross-referencing and finding all changed definitions.

The parameters (all in the compiler package) are:

*produce-xref-info*
When true, the compiler produces information for the Cross Referencer.

*load-xref-info*
When true, the cross-referencing information produced by the compiler is loaded when the corresponding file is loaded.

*notice-changed-definitions*
When true, the Cross Referencer notices when a function is redefined, including an interpreted redefinition.

*source-level-debugging*
When true, the compiler generates information used by the debugger.
toggle-source-debugging modifies the status of the variables, and then returns the new value. To check whether all the variables are set to true, without modifying them, use source-debugging-on-p.

Cross-referencing information is used by the functions who-calls, who-binds, who-references, who-sets, and friends.

Compatibility notes

In LispWorks 4.2 and earlier, toggle-source-debugging controlled source file recording information. In LispWorks 4.3 and later, this is controlled independently by *record-source-files*.

See also

source-debugging-on-p

total-allocation

Function

Summary

Calculate memory consumed since the image was started.

Package

hcl

Signature

total-allocation

Arguments

None.

Values

Returns the amount allocated

Description

This function calculates the total amount of memory consumed since the current image was created. Use at the start and end of a piece of code, to see how much it allocates.

See also

find-object-size

room

“Memory Management in 32-bit LispWorks” on page 135
*traced-arglist*  

Variable

Summary  The list of arguments given to the function being traced.

Package  hcl

Initial value  nil

Description  Upon entering a function that is being traced, *traced-arglist* is bound to the list of arguments given to the function. *traced-arglist* is then printed after the function name in the output from tracing. It is accessible in the :before and :after forms to trace. However, care should be used when manipulating this variable, since it is the value of *traced-arglist* itself that is used when calling the traced function. Thus if this value is altered by the :before forms then the function receives the altered argument list.

Example  

USER 14 > (trace (+ :before

((setq *traced-arglist*(

(mapcar #'1+

*traced-arglist*)))))

+)

USER 15 > (+ 1 2 3)

0 + > (1 2 3)

(2 3 4)

0 + < (9)

9

Notes  *traced-arglist* is an extension to Common Lisp.

See also  trace

*traced-results*  

Variable

Summary  The list of results from the function being traced.
**Package**  
hcl

**Initial value**  
nil

**Description**  
Upon leaving a function that is being traced, *traced-results* is bound to the list of results from the function. *traced-results* is then printed after the function name in the output from tracing. It is accessible in the :after forms to trace. However care should be used when manipulating this variable, since it is the value of *traced-results* itself that is used when returning from the traced function. Thus if this value is altered by the :after forms then the caller of the traced function receives the altered results.

**Example**

```lisp
USER 5 > (trace (ceiling
    :after
    ((setq *traced-results*
      (mapcar #'-1 *traced-results*)))))
CEILING
USER 6 > (multiple-value-call #'+ (ceiling 4 3))
0 CEILING > (4 3)
0 CEILING < (2 -2)
(1 -3)
-2
```

**Notes**

*traced-results* is an extension to Common Lisp.

**See also**

trace

---

**Variable**  

**Summary**  
The amount of extra indentation in the trace output for each level of nesting.

**Package**  
hcl

**Initial value**  
2
Description  

\textit{*trace-indent-width*} is the extra amount by which the traced output for function calls is indented upon entering a deeper level of nesting (that is, a traced call from a function that is itself traced). If it is 0 then no indentation occurs.

Example  

CL-USER 1 > (setq *trace-indent-width* 4  
*max-trace-indent* 50)
50

CL-USER 2 > (defun quad (a b c) (- (* b b) (* 4 a c)))
QUAD

CL-USER 3 > (trace quad *)
(QUAD *)

CL-USER 4 > (quad 4 3 14)
0 QUAD > ...
   >> A : 4
   >> B : 3
   >> C : 14
   1 * > ...
      >> SYSTEM::ARGS : (3 3)
  1 * < ...
      << VALUE-0 : 9
  1 * > ...
      >> SYSTEM::ARGS : (4 4 14)
  1 * < ...
      << VALUE-0 : 224
0 QUAD < ...
   << VALUE-0 : -215
   -215

Notes  

\textit{*trace-indent-width*} is an extension to Common Lisp.

See also  

trace

\textbf{*trace-level*}  

\textit{Variable}

Summary  
The current depth of tracing.

Package  
hcl
**trace-level** is a special variable whose value is the current depth of tracing. The current value of **trace-level** is printed before the function name during the output from tracing.

**Example**

```
USER 8 > (defun fac (n) (if (<= n 1) 1 (* n (fac (1- n)))))
```

```
FAC
USER 9 > (trace fac)
FAC
USER 10 > (fac 3)
0 FAC > (3)
1 FAC > (2)
2 FAC > (1)
2 FAC < (1)
1 FAC < (2)
0 FAC < (6)
6
```

**Notes**

**trace-level** is an extension to Common Lisp.

**See also**

trace

---

**trace-print-circle**

**Variable**

**Summary**

Controls how circular structure are printed in trace output.

**Package**

hcl

**Initial value**

nil

**Description**

**trace-print-circle** controls how circular structures are printed during output from tracing. It allows the printing of circular structures by the tracer to be controlled indepen-
dently of the usual printing mechanism, which is governed by \texttt{*print-circle*}. \texttt{*print-circle*} is bound to the value of \texttt{*trace-print-circle*} while printing tracing information.

\begin{example}

\begin{verbatim}
USER 19 > (setq *trace-print-circle* t)
T
USER 20 > (defun circ (l)
  (rplacd (last l) l)
  l)
CIRC
USER 21 > (trace second)
SECOND
USER 22 > (second (circ '(1 2 3 4)))
 0 SECOND > (#1=(1 2 3 4 . #1#))
 0 SECOND < (2) 2
\end{verbatim}
\end{example}

Notes
\texttt{*trace-print-circle*} is an extension to Common Lisp.

See also
trace

\begin{variable}{*trace-print-length*}

Summary
The number of components of an object that are printed in trace output.

Package
hcl

Initial value
100

Description
\texttt{*trace-print-length*} controls the number of components of an object which are printed during output from tracing. If its value is a positive integer then the first \texttt{*trace-print-length*} components are printed.

\texttt{*print-length*} is bound to the value of \texttt{*trace-print-length*} while printing tracing information. If \texttt{*trace-}
print-length* is nil then all the components of the object are printed.

Example

USER 5 > (trace append)
APPEND
USER 6 > (setq *trace-print-length* 3)
3
USER 7 > (dotimes (i 10) (setq li (if (zerop i)
nil
(cons i li)))
NIL
USER 8 > (append li '(a b))
0 APPEND > ((9 8 7 ...) (A B))
0 APPEND < ((9 8 7 ...))
(9 8 7 6 5 4 3 2 1 A B)

Notes

*trace-print-length* is an extension to Common Lisp.

See also

trace

*trace-print-level* Variable

Summary

The depth to which nested objects are printed in trace output.

Package

hcl

Initial value

5

Description

*trace-print-level* controls the depth to which nested objects are printed during output from tracing. If its value is a positive integer then components at or above that level are suppressed. By definition an object to be printed is considered to be at level 0, its components are at level 1, their sub-components are at level 2, and so on.

*print-level* is bound to the value of *trace-print-level* while printing tracing information. If *trace-
print-level* is nil then objects are printed without regard to depth.

Examples

USER 8 > (trace append)
APPEND
USER 9 > (dotimes (i 10) (setq li (if (zerop i)
nil
(list i li))))
NIL
USER 10 > (append li '(a b))
0 APPEND > ((9 (8 (7 (6 #)))) (A B))
0 APPEND < ((9 (8 (7 (6 #))) A B))
(9 (8 (7 (6 (5 (4 (3 (2 (1 NIL)))))))) A B)

Notes
*trace-print-level* is an extension to Common Lisp.

See also
trace

*trace-print-pretty*  
Variable

Summary
Controls the amount of whitespace in trace output.

Package
hcl

Initial value
nil

Description
*trace-print-pretty* controls the amount of whitespace printed during output from tracing. If it is not nil then extra whitespace is inserted to make the output more comprehensible. *print-pretty* is bound to the value of *trace-print-pretty* while printing tracing information.

Examples

USER 6 > (trace macroexpand-1)
MACROEXPAND-1
USER 7 > (setq *trace-print-pretty* t
         *print-pretty* nil)
NIL
USER 8 > (defmacro sum (n)
    '(do ((i 0 (1+ i))
        (res 0 (+ i res)))
    ((= i ,n) res)))
SUM
USER 9 > (macroexpand-1 '(sum 3))
0 MACROEXPAND-1 > ((SUM 3))
0 MACROEXPAND-1 < ((DO ((I 0 (1+ I))
        (RES 0 (+ I RES)))
    ((= I 3)
        RES))
    T)
(DO ((I 0 (1+ I)) (RES 0 (+ I RES))) ((= I 3) RES))
T

Notes *trace-print-pretty* is an extension to Common Lisp.

See also trace

*trace-verbose*  

Variable
Summary Controls how arguments and values are printed in trace output.
Package hcl
Initial value :only
Description *trace-verbose* controls the way arguments and values are printed in trace output.
If the value is not nil then trace attempts to decode the arguments and values, and prints them.
When the value is :only, trace does not print the lists of arguments and values after the function name.
Notes *trace-verbose* is an extension to Common Lisp.
try-compact-in-generation  

Function

Summary
Compacts the most fragmented segment(s) in a generation in 32-bit LispWorks.

Package
hcl

Signature
try-compact-in-generation generation-number time-threshold
&optional fraction-threshold => result

Arguments
generation-number
0 for the most recent generation, 1 for the most recent two generations, and so on up to a maximum (usually 3). Numbers outside this range signal an error.

time-threshold A real number.

fraction-threshold
A real number between 0 and 1, defining the minimum fragmentation to actually compact. The default is 0.25.

Values
result A boolean.

Description
try-compact-in-generation finds the most fragmented segment in the generation specified. If time-threshold is positive, it compacts this segment, and repeats this operation until time-threshold seconds have elapsed. At this point try-compact-in-generation returns, with value t if at least one segment was compacted and value nil otherwise. Because the operation cannot be stopped in the middle, the actual time taken will always be larger than time-threshold.

If fraction-threshold is 1, try-compact-in-generation does nothing. If fraction-threshold is 0,
**try-compact-in-generation** will compact all uncompacted segments (unless it runs out of time). With the default (0.25) **try-compact-in-generation** compacts only moderately fragmented segments.

If **time-threshold** is negative, then **try-compact-in-generation** does not actually compact any segments. **result** is a boolean indicating whether **try-compact-in-generation** would actually try to compact a segment if it were to be called with a positive **time-threshold** and the other arguments unchanged.

This function is typically used after a call to **check-fragmentation**. For more information, see “Controlling Fragmentation” on page 144.

**Notes**

try-compact-in-generation is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations, where marking-gc with the what-to-copy argument offers similar functionality (although set-blocking-gen-num is intended to solve the problem of fragmentation automatically).

**See also**

check-fragmentation
try-move-in-generation
“Memory Management in 32-bit LispWorks” on page 135

---

**try-move-in-generation**

**Summary**

Moves objects out of the most fragmented segment(s) in a generation, leaving them empty in 32-bit LispWorks.

**Package**

hcl

**Signature**

try-move-in-generation generation-number time-threshold &optional fraction-threshold => result
Arguments


generation-number
0 for the most recent generation, 1 for the most recent two generations, and so on up to a maximum (usually 3). Numbers outside this range signal an error.

time-threshold
A real number.

fraction-threshold
A real number between 0 and 1, defining the minimum fragmentation to actually move. The default is 0.25.

Values

result
A boolean.

Description

try-move-in-generation finds the most fragmented segment in the generation specified. If time-threshold is positive, it moves objects out of this segment, leaving it empty, and repeats this operation until time-threshold seconds have elapsed. At this point try-move-in-generation returns, with value \texttt{t} if at least one segment was moved and value \texttt{nil} otherwise. Because the operation cannot be stopped in the middle, the actual time taken will always be larger than time-threshold.

If fraction-threshold is 1, try-move-in-generation does nothing. If fraction-threshold is 0, try-move-in-generation will move all uncompacted segments (unless it runs out of time). With the default (0.25) try-move-in-generation moves only moderately fragmented segments.

If time-threshold is negative, then try-move-in-generation does not actually move any segments. result is a boolean indicating whether try-move-in-generation would actually try to move a segment if it were to be called with a positive time-threshold and the other arguments unchanged.

This function is typically used after a call to check-fragmentation. For more information, see “Controlling Fragmentation” on page 144.
try-move-in-generation is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations, where marking-gc with the what-to-copy argument offers similar functionality (although set-blocking-gen-num is intended to solve the problem of fragmentation automatically).

See also
check-fragmentation
try-compact-in-generation
“Guidance for control of the memory management system” on page 130

### undefine-declaration

**Function**

**Summary**
Remove a user declaration handler.

**Package**
hcl

**Signature**
undefine-declaration decl-name => decl-name

**Arguments**

- **decl-name**
  A symbol.

**Values**

- **decl-name**
  A symbol.

**Description**
The function **undefine-declaration** causes decl-name not to be recognized as a declaration. It can be used to undo the effect of **define-declaration**, but also to undo the effect of proclaiming decl-name as a declaration by **proclaim** or **declaim**. Note that **undefine-declaration** is a function, unlike **define-declaration** which is a macro.

**See also**

- define-declaration
- declare
- declaration-information
- function-information
unlocked-queue

Type

Summary
A fast queue.

Package
hcl

Description
The type **unlocked-queue** is a fast queue (first in, first out) that is unlocked, not thread-safe and does not have waiting functionality. It does not do anything that **mailbox** cannot do, but it is faster. It is useful when you always access the queue together with other operations that need to be "atomic", so that you need a lock around them anyway, or when queueing and de-queueing is done on the same process.

See also
**make-unlocked-queue**

unwind-protect-blocking-interrupts

Macro

Summary
Does **unwind-protect** blocking interrupts.

Package
hcl

Signature
unwind-protect-blocking-interrupts protected-form &rest
  cleanups => results

Arguments
protected-form A form.

  cleanups Forms.

Values
results The values of protected-form.
The macro `unwind-protect-blocking-interrupts` executes `protected-form` with interrupts blocked. On exit, whether local or not, the cleanups are executed with interrupts blocked.

In compiled code, the macro is equivalent to

```lisp
(mp:with-interrupts-blocked
 (unwind-protect
   protected-form
   (mp:current-process-block-interrupts)
   cleanup1 cleanup2 ...))
```

However, in interpreted code the macro is expanded to ensure that the call to `(mp:current-process-block-interrupts)` actually happens. If the above form is interpreted and `protected-form` uses `current-process-unblock-interrupts`, the evaluator may throw (if the process is killed, for example) before calling `current-process-unblock-interrupts`.

### Notes

1. Both the protected form and the cleanups can block and unblock interrupts using `current-process-block-interrupts` and `current-process-unblock-interrupts`. Typically the protected form would set up something and then unblock the interrupts. The cleanups may unblock interrupts if some of the cleanups are essential and others are not.

2. Blocking interrupts causes the process to not respond to interrupts, including killing. You should make sure that forms which are executed with interrupts blocked do not hang.

### See also

- `current-process-block-interrupts`
- `current-process-unblock-interrupts`
- `unwind-protect-blocking-interrupts-in-cleanups`
unwind-protect-blocking-interrupts-in-cleanups

Macro

Summary
Does unwind-protect blocking interrupts around the cleanups.

Package
hcl

Signature
unwind-protect-blocking-interrupts-in-cleanups protected-form &rest cleanups => results

Arguments
protected-form A form.
cleanups Forms.

Values
results The values of protected-form.

Description
The macro unwind-protect-blocking-interrupts-in-cleanups executes protected-form. On exit, whether local or not, the cleanups are executed with interrupts blocked.

In compiled code, the macro is equivalent to

(unwind-protect
 protected-form
 (mp:with-interrupts-blocked cleanup1 cleanup2 ..))

However, in interpreted code the macro is expanded to ensure that the body of mp:with-interrupts-blocked actually happens. If the form above is interpreted the evaluator may throw (if the process is killed, for example) before completing macroexpansion of mp:with-interrupts-blocked and doing the actual blocking.

Notes
1. cleanups can block and unblock interrupts using current-process-block-interrupts and current-process-unblock-interrupts. This may be useful if some of the cleanups are essential and others are not.
2. Blocking interrupts causes the process to not respond to interrupts, including killing. You should make sure that forms which are executed with interrupts blocked do not hang.

See also

- current-process-block-interrupts
- current-process-unblock-interrupts
- unwind-protect-blocking-interrupts
- with-interrupts-blocked

**variable-information**

*Function*

**Summary**

Returns information about the variable bindings of a symbol in an environment.

**Package**

hcl

**Signature**

variable-information variable &optional env => kind, localp, decls

**Arguments**

- variable
  - A symbol
- env
  - An environment or nil

**Values**

- kind
  - nil or one of the keywords :special, :lexical, :symbol-macro and :constant.
- localp
  - A boolean
- decls
  - An a-list

**Description**

The function `variable-information` returns information about how the variable `symbol` is bound in the environment `env`.

The value of `kind` will be as follows:

- nil
  - There is no information about `variable` in `env`
- :special
  - `variable` has a special binding in `env`
:lexical  \( \text{variable} \) has a lexical binding in \( \text{env} \)
:symbol-macro  \( \text{variable} \) has a symbol-macro binding in \( \text{env} \)
:constant  \( \text{variable} \) has a constant binding in \( \text{env} \)

\( \text{localp} \) will be true if \( \text{variable} \) is bound by a form that has lexical scope (for example \texttt{let, lambda}) or false if \( \text{variable} \) has global scope (for example \texttt{defvar}).

decls is an a-list of declarations that refer to \( \text{variable} \). The \texttt{cdr} of each pair is specified according to the \texttt{car} of the pair as follows:

dynamic-extent
\( \text{The cdr} \) is non-nil if \( \text{variable} \) is declared \texttt{dynamic-extent} in \( \text{env} \).

ignore
\( \text{The cdr} \) is non-nil if \( \text{variable} \) is declared \texttt{ignore} in \( \text{env} \).

type
\( \text{The cdr} \) is the type specifier that is declared for \( \text{variable} \) in \( \text{env} \) if any.

Notes
\( \text{variable-information} \) is part of the environment access API which is based on that specified in \textit{Common Lisp: the Language (2nd Edition)}.

See also
augment-environment
declaration-information
define-declaration
function-information
map-environment

who-binds

Function

Summary
Returns the definitions which bind a special variable.

Package  hcl
The HCL Package

**Signature**  
who-binds symbol => result

**Arguments**  
symbol A special variable.

**Values**  
result A list.

**Description**  
The function who-binds returns a list of dspecs naming the definitions which bind the special variable symbol.

**Notes**  
The cross-referencing information used by who-binds is generated when code is compiled with source-level debugging switched on.

**See also**  
binds-who  
toggle-source-debugging  
who-sets  
who-references

---

**who-calls**  

**Function**

**Summary**  
Returns the callers of a function.

**Package**  
hcl

**Signature**  
who-calls dspec => callers

**Arguments**  
dspec A dspec.

**Values**  
callers A list.

**Description**  
The function who-calls returns a list of dspecs naming the definitions which call the function named by dspec.

See also the editor commands List Callers and Show Paths To.
Notes  The cross-referencing information used by who-calls is generated when code is compiled with source-level debugging switched on.

See also  calls-who
toggle-source-debugging

who-references  

Function
Summary  Returns the definitions which reference a special variable.

Package  hcl

Signature  who-references symbol => result

Arguments  symbol  A special variable.

Values  result  A list.

Description  The function who-references returns a list of dspecs naming the definitions which reference the special variable symbol.

Notes  The cross-referencing information used by who-references is generated when code is compiled with source-level debugging switched on.

See also  references-who
toggle-source-debugging
who-binds
who-sets

who-sets  

Function
Summary  Returns the definitions which set a special variable.
The HCL Package

Package: hcl

Signature: who-sets symbol => result

Arguments:
- symbol: A special variable.

Values:
- result: A list.

Description: The function who-sets returns a list of dspecs naming the definitions which set the value of the special variable symbol.

Notes: The cross-referencing information used by who-sets is generated when code is compiled with source-level debugging switched on.

See also:
- sets-who
- toggle-source-debugging
- who-binds
- who-references

with-code-coverage-generation

Macro

Summary: Switches code coverage generation during the execution of a body of code.

Package: hcl

Signature: with-code-coverage-generation (&key on atomic-p counters force count-implicit-branch) &body body => result

Arguments:
- on: A boolean.
- atomic-p: A boolean.
- counters: A boolean.
- force: A boolean.
- count-implicit-branch

with-code-coverage-generation

Macro

Summary: Switches code coverage generation during the execution of a body of code.

Package: hcl

Signature: with-code-coverage-generation (&key on atomic-p counters force count-implicit-branch) &body body => result

Arguments:
- on: A boolean.
- atomic-p: A boolean.
- counters: A boolean.
- force: A boolean.
- count-implicit-branch
A boolean.

Values

\[ \text{result} \]

The result of executing \textit{body}.

Description

The macro \texttt{with-code-coverage-generation} switches code coverage generation on or off inside the dynamic scope of \textit{body}.

The arguments are interpreted as by \texttt{generate-code-coverage}.

See also \texttt{generate-code-coverage}

\textbf{with-ensuring-gethash}

\textit{Macro}

Summary

A thread-safe way to get a value from a \texttt{hash-table}, adding a value if the key is not present. Allows a complicated form to construct the new value.

Package \texttt{hcl}

Signature

\texttt{with-ensuring-gethash key hash-table &key constructor constructor-form in-lock-constructor in-lock-constructor-form => result}

Arguments

\begin{itemize}
  \item \texttt{key} \hspace{1cm} A Lisp object.
  \item \texttt{hash-table} \hspace{1cm} A \texttt{hash-table}.
  \item \texttt{constructor} \hspace{1cm} A function designator for a function of no arguments, or \texttt{nil}.
  \item \texttt{constructor-form} \hspace{1cm} A Lisp form, or \texttt{nil}.
  \item \texttt{in-lock-constructor} \hspace{1cm} A function designator for a function of one argument, or \texttt{nil}.
  \item \texttt{in-lock-constructor-form}
\end{itemize}
A Lisp form, or nil.

Values

<table>
<thead>
<tr>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>result</td>
</tr>
</tbody>
</table>

Description

The macro `with-ensuring-gethash` gets the value for the key `key` from the hash table `hash-table`, and if this fails constructs a new value, puts it in the table and returns it. `with-ensuring-gethash` does this in a thread-safe way, which means that all threads calling it with the same `key` and `hash-table` return the same value (as long as nothing removes it from the table).

Only one of `constructor-form` or `constructor` can be non-nil. When `key` is not found, `constructor-form` or `constructor` is used to construct the new value. If `constructor` is non-nil, it is called without arguments. If `constructor-form` is non-nil, it is executed. If both are nil, the new value is nil unless the `in-lockconstructor` or `in-lock-constructor-form` construct it. The call or execution of the `constructor` or `constructor-form` is done without any lock. The result may be discarded if, by the time it returned, there is a match for the key in the table.

Only one of `in-lock-constructor` or `in-lock-constructor-form` can be non-nil, which is used when the key is not found after constructing the new value. If `in-lock-constructor-form` is non-nil, it is executed and the result is the actual value to use (the result of the construction by `constructor-form` or `constructor` is ignored). If `in-lock-constructor` is non-nil, it is called with the result of the construction by `constructor-form` or `constructor`, and the result is used as the new value. In either case, the call or execution is done with `hash-table` locked, and the result is guaranteed to be put in `hash-table` and returned. If both `in-lock-constructor` and `in-lock-constructor-form` are nil, the result of the construction is used.

Notes

1. When both `constructor-form` and `in-lock-constructor-form` are nil, `gethash-ensuring` is probably simpler and better.
2. In most situations, doing all the construction out of the lock is better than doing anything inside the lock. It means that sometimes the work that was done in the constructions is wasted because another thread put the value in the table, but that overhead is normally less significant than the overhead of holding the lock for longer, with the associated potential deadlocks.

See also

- `ensure-hash-entry`
- `gethash-ensuring`
- “Modifying a hash table with multiprocessing” on page 279

---

**with-hash-table-locked**

*Macro*

**Summary**

Evaluates code with a hash-table locked against modification by other threads.

**Package**

*hcl*

**Signature**

`with-hash-table-locked hash-table &body body => results`

**Arguments**

- `hash-table`: A hash table.
- `body`: Forms.

**Values**

- `results`: The results of evaluating `body`.

**Description**

The macro *with-hash-table-locked* evaluates `body` with the hash table `hash-table` locked against modification by other threads. The current thread can modify `hash-table`. *with-hash-table-locked* is useful not only for multiple accesses to the same table, but also when an access to the table must be consistent with some other operation, avoiding the need to make a separate lock,
See also  
gethash-ensuring
make-hash-table
modify-hash
with-ensuring-gethash
“Atomicity and thread-safety of the LispWorks implementation” on page 265

with-heavy-allocation  
Macro

Summary  
Slows up garbage collection during the execution of code that allocates a lot of space.

Package  
hcl

Signature  
with-heavy-allocation &rest body => result

Arguments  
body  
The forms for which you want the garbage collector to behave differently from normal.

Values  
result  
The result of executing body.

Description  
The macro with-heavy-allocation is for use with code that allocates a lot of space but is not interactive. It ensures that garbage collection (GC) is carried out less frequently while these forms are being executed. However, each GC may take longer.

Compatibility notes  
In LispWorks 5.0 with-heavy-allocation is implemented only in 32-bit LispWorks. In version 5.1 and later it is implemented in 64-bit LispWorks as well.

See also  
avoid-gc
gc-if-needed
get-gc-parameters
mark-and-sweep
normal-gc
**set-gc-parameters**

**finish-heavy-allocation**

**without-interrupts**

“Memory Management in 32-bit LispWorks” on page 135

---

**with-output-to-fasl-file**

**dump-form**

**dump-forms-to-file**

---

### Macro and Functions

**Summary**

Dump forms to a file in a binary format, which can then be loaded using `load-data-file`.

**Package**

`hcl`

**Signature**

```lisp
(with-output-to-fasl-file (fast-stream-var pathname &key overwrite dump-standard-objects delete-on-error) &body body) => nil
dump-form form fasl-stream => nil
dump-forms-to-file pathname forms &key overwrite dump-standard-objects delete-on-error => nil
```

**Arguments**

- `fast-stream-var` A variable.
- `pathname` A pathname designator.
- `overwrite` A boolean.
- `dump-standard-objects` A boolean.
- `delete-on-error` A boolean.
- `body` Lisp code that calls `dump-form`.
- `form` A form.
- `fast-stream` The value of `fast-stream-var`.
- `forms` A list of forms.
Values

Each of these macros and functions returns `nil`.

Description

The macro `with-output-to-fasl-file` and the functions `dump-form` and `dump-forms-to-file` allow you to dump forms to a file in a binary format, which can then be loaded using `load-data-file`.

They provide these two ways to achieve the same thing:

- `dump-forms-to-file` simply dumps the forms in `forms` into the file specified by `pathname`.

- `with-output-to-fasl-file` binds `fasl-stream-var` to an opaque structure associated with `pathname`. Inside the `body`, `dump-form` is used to dump individual forms to the file.

`overwrite` specifies what to do if `pathname` already exists. If `overwrite` is non-nil, `dump-forms-to-file` and `with-output-to-fasl-file` overwrite the existing file, otherwise they signal an error. The default value of `overwrite` is `t`.

`delete-on-error` specifies what to do in case of a non-local exit from the `body` of `with-output-to-fasl-file` or from `dump-forms-to-file` (typically abort after an error). By default, the file is deleted, but if `delete-on-error` is `nil` then the file is left as it is. The default value of `delete-on-error` is `t`.

`dump-standard-objects` specifies what to do when trying to dump a standard object (that is, an instance of a subclass of `standard-object`) which does not have a user-defined `make-load-form`. If `dump-standard-objects` is `nil`, an error is signaled. If `dump-standard-objects` is non-nil, the instance is dumped using `make-load-form-saving-slots`. The default value of `dump-standard-objects` is `nil`.

When the generated file is loaded by `load-data-file`, the forms are loaded and by default evaluated, though `load-data-file` can also load without evaluating. If `callback` is passed to `load-data-file`, it gets each of the results. Otherwise the results are discarded (except being printed when
passing :print t). Hence to be useful, either load-data-file must be called with callback, or evaluation of the forms should have some side effect, for example setting the value of some special symbol or adding entries to some global table.

For a form which is not a list or an object with make-load-form, or is a quoted list, eval does nothing. Dumping such forms and then using using the callback in load-data-file to do some work with them is the natural way of using dump-forms-to-file or with-output-to-fasl-file and load-data-file to transfer large amounts of data.

Files generated by dump-forms-to-file or with-output-to-fasl-file can be loaded (by load-data-file) on any LispWorks platform with the same byte order. All x86/x64 architectures have the same byte order (little-endian), so load-data-file on any x86/x64 architecture can be used to load a data file that was generated on any x86/x64 architecture. The ARM architectures have the same byte order as x86/x64. The reverse byte order (big-endian) is used by AIX and SPARC (old Solaris).

Notes

1. The dumping of objects is done the same way that compile-file dumps when it creates a fasl file, except for the treatment of standard objects when dump-standard-objects is non-nil.

2. Dumping means creating a deep copy of the form. The elements and slots of lists, arrays of element type t, structures (unless they have a make-load-form), and, when dump-standard-objects is non-nil, standard objects without make-load-form are dumped recursively.

3. dump-forms-to-file and with-output-to-fasl-file cope with cyclic structures.

4. If you want to dump parts of cyclic structures, you can stop the recursion by defining an appropriate make-load-form method for the objects at the nodes where the recursion should stop.
5. A fasl file created using `with-output-to-fasl-file` must be loaded only by `load-data-file`, and not by `load`.

Example

```lisp
(dump-forms-to-file "my-forms.data"
  '(#(1 2 3)
    89
    (* 7 7)
    '(* 9 9)))
```

Note that the first * form lacks a quote while the second has a quote.

Then (potentially in a different LispWorks version and/or on a different architecture) this:

```lisp
(load-data-file "my-forms.data" :callback 'print)
```

prints this:

```
#(1 2 3)
89
49
(* 9 9)
```

In contrast, loading the same binary file without evaluation:

```lisp
(load-data-file "my-forms.data" :callback 'print :eval nil)
```

prints this:

```
#(1 2 3)
89
(* 7 7)
(QQUOTE (* 9 9))
```

See also `load-data-file`
with-ring-locked  

Macro

Summary  
Locks a ring such that no other thread can access it while some code is executed.

Package  
hcl

Signature  

\[
\text{with-ring-locked (ring \&optional whostate timeout) \&body body} \Rightarrow \text{result}
\]

Arguments  

\begin{itemize}
  \item \text{ring}  
  A ring object created by \text{make-ring}.
  \item \text{whostate}  
  The status of the process while the ring is locked.
  \item \text{timeout}  
  A timeout period, in seconds.
  \item \text{body}  
  Lisp code.
\end{itemize}

Values  

\text{result}  
The result of executing \text{body}.

Description  
The macro \text{with-ring-locked} locks the ring \text{ring} that during the execution of \text{body} no other thread can access ring, whether for modification or merely reading values.  
\text{whostate} and \text{timeout} are used in the same way as in \text{with-lock}.

See also  
\text{make-ring}

without-code-coverage  

Macro

Summary  
Prevents generation of code coverage for a body of code.

Package  
hcl

Signature  

\[
\text{without-code-coverage \&body body} \Rightarrow \text{result}
\]

Arguments  

\text{body}  
Lisp forms.
Values
\[ \text{result} \]
The result of evaluating \( \text{body} \).

Description
The macro \texttt{without-code-coverage} prevents generation of code coverage for the forms of \( \text{body} \).

\( \text{body} \) is evaluated as an implicit \texttt{progn}, except that inside \( \text{body} \) the compiler does not generate code coverage counters, unless \texttt{force} was supplied non-nil to \texttt{generate-code-coverage} or \texttt{with-code-coverage-generation}.

\texttt{without-code-coverage} is useful for error forms that you do not want to be counted.

Notes
There will be a counter for the (\texttt{without-code-coverage} ...) form itself. If you do not want this counter, use \texttt{error-situation-forms} instead.

See also
\texttt{error-situation-forms}
\texttt{generate-code-coverage}
\texttt{with-code-coverage-generation}

\textbf{write-string-with-properties} \hspace{1cm} \textit{Function}

Summary
Writes the string to the stream, and adds properties if the stream supports it.

Package
hcl

Signature
\texttt{write-string-with-properties} \texttt{string properties &optional stream &key start end => string}

Arguments
\begin{itemize}
  \item \texttt{string} \hspace{1cm} A string.
  \item \texttt{properties} \hspace{1cm} A property list.
  \item \texttt{stream} \hspace{1cm} An output stream designator.
\end{itemize}
**start, end**

Bounding index designators of string.

**Values**

| string | A string. |

**Description**

The function `write-string-with-properties` writes a part of string, bounded by `start` and `end`, to `stream` exactly like `cl:write-string` does, and then adds properties if `stream` supports this operation. Currently the only types of `stream` that support properties are Editor streams, which means streams that write to Editor buffers.

`properties` must be a property list, with alternating `key` and `value`. When adding the properties, each pair of `key` and `value` is processed separately, in the order they occur in the list.

If `key` is `:highlight`, then `value` must specify a `editor:face`. It can be an `editor:face` object, a face name (which means a symbol that was used as the name in `editor:make-face`), or one of the following keywords:

- `:underline` Underline the text.
- `:bold` Make the text bold.
- `:italic` Make the text italic.
- `:bold-italic` Make the text bold and italic.
- `:inverse` Invert the background and foreground.
- `:compiler-error-highlight`
  or `:compiler-warning-highlight`
  or `:compiler-note-highlight`
  The faces that the compiler use when it outputs to an editor buffer.
- `:editor-error-highlight`
  The face that the editor uses in the echo area when it reports an editor error.
If `key` is :menu-items, `value` specifies menu items that are added to the context menu that is displayed when the current point is inside `string` in the Editor buffer. The value must be a list where each element specifies a menu-item by a list of three elements: `title`, `function` and `arg`. `title` must be a string, and is what the end user sees. If the user selects this item, `function` is called with `arg` as a single argument.

If `key` is not :menu-items or :highlight, the `key` / `value` is added as a text property using `editor:put-text-property`, with `key` as the `property` argument and `value` as `value`.

**Example**

Output "A string in bold" in bold face to `mp:*background-standard-output*`:

```lisp
(write-string-with-properties
 "A string in bold"
 '(:highlight :bold)
 mp:*background-standard-output*)
```

**Note**

`write-string-with-properties` can be used where ever `write-string` is used, because it does exactly the same for streams that do not implement properties.

The LispWorks compiler uses `write-string-with-properties` to write compiler errors, warnings and notes with :complier-...-highlight keywords above. This is how it produces the colored errors/warnings/notes when it writes to the Listener or is invoked in the Editor.

Support for :menu-items is implemented by the method:

```lisp
(method capi:pane-popup-menu-items
  (capi:editor-pane capi:interface))
```

Therefore, if this method is not called, for example if you make a `capi:editor-pane` and pass it :pane-menu, then items will not be added to the menu.

**See also**

`cl:write-string`
This chapter describes the symbols in the **LINK-LOAD** package.

**Note:** this chapter applies only to LispWorks for SPARC Solaris.

### break-on-unresolved-functions

**Function**

<table>
<thead>
<tr>
<th>Package</th>
<th>link-load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>break-on-unresolved-functions &amp;optional stream</td>
</tr>
<tr>
<td>Arguments</td>
<td>stream An output stream for message reporting. If set to nil, then no output will be produced. By default this is t.</td>
</tr>
<tr>
<td>Description</td>
<td>The function break-on-unresolved-functions produces break-on-entry code for all currently undefined but referenced (that is, unresolved) foreign symbols, so that if an undefined foreign function is called from within the foreign code, a Lisp error will occur. Break-on-entry code will also be produced for any new unresolved symbols loaded later in your Lisp session.</td>
</tr>
</tbody>
</table>
The special variable `foreign:*break-on-unresolved-functions*` will, when set to non-nil, produce break-on-entry code for all new unresolved symbols that are loaded, but won’t do so for symbols already loaded. By default this variable is set to `nil`.

See also `read-foreign-modules`

### foreign-symbol-address

**Function**

**Package** link-load

**Signature** `foreign-symbol-address name &key errorp functionp => result`

**Arguments**

- `name`: The name of a foreign symbol.
- `errorp`: A boolean.
- `functionp`: A boolean.

**Values**

- `result`: The address of `name` or `nil`.

**Description**

The function `foreign-symbol-address` is used to find out whether a foreign symbol is defined, by looking for it in the foreign-symbol table. If its associated object code has been loaded into the image, its address is returned. Otherwise `nil` is returned, unless `errorp` is `nil`.

The `errorp` keyword defines the behavior of the function when a symbol has not been defined. If it is non-nil (which is the default value), then an error will be signaled. If it is `nil`, no error will be reported, and the function will return `nil`.

The `functionp` keyword is used to specify the kind of symbol sought. If it is `t`, `foreign-symbol-address` will assume that `name` is the name of a function. If it is `nil` it will assume that `name` is the name of a variable. The default value is `t`. 
Example

(foreign-symbol-address 'chmod)

See also

get-foreign-symbol

get-foreign-symbol

Function

Package

link-load

Signature

get-foreign-symbol name &optional force => result

Arguments

name A symbol or string.

force A keyword.

Values

result A foreign symbol.

Description

This function gets a foreign symbol or it may be used to explicitly register an undefined symbol.

name is a symbol or string to look up or to create as a foreign symbol. If it is a symbol, the symbol looked for is that which the function lisp-name-to-foreign-name would produce. If name is a string, it is taken literally.

If supplied and the symbol is not already defined as a foreign symbol, force forces it to be an undefined foreign symbol. This provides a reference to the symbol so that a subsequent call to read-foreign-modules will attempt to resolve it.

Example

(get-foreign-symbol 'my-func-not-yet-loaded t)

Notes

It is not usually necessary to use this function. In order to examine whether a foreign symbol is defined, use foreign-symbol-address. The act of defining a foreign function using fli:define-foreign-function makes the symbol undefined, so the use of force is not usually needed.
See also  
- foreign-symbol-address
- lisp-name-to-foreign-name
- read-foreign-modules

### lisp-name-to-foreign-name

**Function**

**Package**  
link-load

**Signature**  

```lisp
lisp-name-to-foreign-name name &key language
```

**Arguments**

- `name`  
  A symbol representing a Lisp name. (Strings are passed unchanged through the function.)

- `language`  
  If :C then an equivalent 'C' name is produced. :FORTRAN is an alternative.

**Description**  
This function provides an equivalent foreign name for a Lisp name, depending on the keyword language.

**Values**  
A string is returned which is a foreign equivalent of the Lisp name supplied. If name is a string, the function returns the string unchanged. If language is a symbol, the 'C' version replaces occurrences of '-' with '_' and adds a leading underscore. The Fortran version replaces occurrences of '-' with '_' and adds a leading and trailing underscore.

**Example**

```lisp
(lisp-name-to-foreign-name 'lisp-name-with-hyphens)
"_lisp_name_with_hyphens"
```

See also  
- get-foreign-symbol

### read-foreign-modules

**Function**

**Package**  
link-load
Signature

read-foreign-modules &rest module-names => t

Arguments

module-names A sequence of strings or pathnames.

Values

t

Description

The function read-foreign-modules reads object files of various formats into the Lisp image. Unresolved references are resolved wherever possible and the names of the foreign functions are made available to the Lisp for direct calling from the Lisp if desired. With no argument, read-foreign-modules scans the default libraries looking for definitions of referenced but undefined symbols.

The module-names argument is a sequence of items representing object files to be loaded. The items may be of type string or pathname, and will be used to look up a corresponding file in the file system. The only exception is if an item is a string beginning "-l" in which case the rest of the string is used to look up a library file using format strings constructed from the values of the variable *default-library-name-search-paths*, the environment variable LD_LIBRARY_PATH and the variable *default-library-names*. Object files of various formats and library files can be handled by read-foreign-modules.

Example

(read-foreign-modules "/usr/users/clc/projects/head.o" 
    "-lW")

Notes

The function read-foreign-modules actually adds the module-names to the list of modules in the variable *default-libraries* and then tries to resolve any undefined symbols using this list. The function get-foreign-symbol may be called to explicitly force a symbol onto the undefined list or the act of defining a foreign function (fli:define-foreign-function) will do it implicitly.
read-foreign-modules may be called at any time during the running of a program and a particular object file may be loaded as often as is necessary.

A warning of any new unresolved references will be printed out after the reading has finished if the flag *unresolved-messages* is set to t (the default is nil). By default messages are printed out about which object modules are being loaded. This may be switched off by setting *coff-loading-verbose* to nil.

See also

get-foreign-symbol
The LISPWORKS Package

This chapter describes symbols available in the LISPWORKS package. This package is used by default. Its symbols are visible in the CL-USER package. Various uses of the symbols documented here are discussed throughout this manual.

**8-bit-string**

<table>
<thead>
<tr>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
</tr>
<tr>
<td>Package</td>
</tr>
<tr>
<td>Signature</td>
</tr>
<tr>
<td>Arguments</td>
</tr>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Description</td>
</tr>
</tbody>
</table>
### 16-bit-string

**Type**

**Summary**
The 16 bit string type.

**Package**
lispworks

**Signature**
16-bit-string length

**Arguments**
length The length of the string (or *, meaning any).

**Description**
The type of strings that can hold simple chars of codes 0…65533. This is the string type that is guaranteed to always take 16 bits per element.

### append-file

**Function**

**Summary**
Appends the contents of a file to another file.

**Package**
lispworks

**Signature**
append-file from to

**Arguments**
from A pathname designator.
to A pathname designator.

**Description**
The function append-file appends the contents of the file from to another file. The file from must exist.

append-file opens from for input and to for output using if-exists :append (see cl:open in the Common Lisp HyperSpec) and copies the contents from from to to.

On any failure append-file signals an error.

append-file does not return a useful value.

**See also**
copy-file
**appendf**

Macro

Summary: Appends lists to the end of a given list.

Package: lispworks

Signature: `appendf place &rest lists => result`

Arguments:
- `place`: A place.
- `lists`: A set of lists.

Values:
- `result`: An object.

Description: The modify macro `appendf` appends the lists given by `lists` to the end of the list in `place`. See `append` for more details.

See also: `removef`

---

**/autoload-asdf-integration**

Variable

Summary: Determines whether ASDF integration code is loaded automatically.

Signature: `autoload-asdf-integration`

Package: lispworks

Initial value: `t`

Description: The variable `autoload-asdf-integration` is consulted used when the LispWorks IDE starts. If its value is true, then the system arranges for ASDF integration code to be loaded automatically when ASDF is loaded.

The ASDF integration code makes the LispWorks IDE tools (System Browser, Search Files) work with ASDF systems.
(defined with `asdf:defsystem`) as well as 'native' systems defined with `defsystem`.

See “Using ASDF” on page 311 for more information about using ASDF with LispWorks.

### base-character

<table>
<thead>
<tr>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
</tr>
<tr>
<td><strong>Package</strong></td>
</tr>
<tr>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>See also</strong></td>
</tr>
</tbody>
</table>

### base-character-p

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
</tr>
<tr>
<td><strong>Package</strong></td>
</tr>
<tr>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
</tr>
<tr>
<td><strong>Values</strong></td>
</tr>
</tbody>
</table>
The function **base-character-p** is a predicate for base characters. 

\( \text{result} \) is \( t \) if \( \text{object} \) is a base character, and \( \text{nil} \) otherwise.

**See also** base-character

---

**base-char-p**

**Function**

**Summary**

A predicate for base characters.

**Package**

lispworks

**Signature**

\[ \text{base-char-p} \ \text{object} \Rightarrow \text{result} \]

**Arguments**

\( \text{object} \) The object to be tested.

**Values**

\( \text{result} \) A boolean.

**Description**

The function **base-char-p** is a predicate for base characters, only with standard spelling.

\( \text{result} \) is \( t \) if \( \text{object} \) is a base character, and \( \text{nil} \) otherwise.

**See also** base-character-p

---

**base-char-code-limit**

**Constant**

**Summary**

Upper bound for character codes in base characters.

**Package**

lispworks

**Description**

The upper exclusive bound for values of \( \text{char-code char} \) among base characters.
The LISPWORKS Package

base-string-p
simple-base-string-p

Summary
The predicates for base strings.

Package
lispworks

Signature
base-string-p object => result
simple-base-string-p object => result

Arguments
object The object to be tested.

Values
result A boolean.

Description
The functions base-string-p and simple-base-string-p are the predicates for base strings and simple base strings respectively.

result is t if object is a base-string (or simple-base-string), and nil otherwise.

See also
base-string
simple-base-string

bmp-char

Summary
The type of characters that fit in the Basic Multilingual Plane.

Package
lispworks

Description
The type bmp-char is the type of characters that fit in the Unicode Basic Multilingual Plane, that is all characters that fit in 16 bits.

The Basic Multilingual Plane (BMP) is the range of Unicode code points below #x10000.
Notes

1. Normally you should not be able to produce a Lisp character object corresponding to a surrogate code point. If such an object is created, it is treated as \texttt{bmp-char}.

2. The corresponding string types are \texttt{bmp-string} and \texttt{simple-bmp-string}. \texttt{bmp-char} can be written to a stream or passed to the FLI with external format \texttt{:bmp} without ever getting an error.

Compatibility note

\texttt{bmp-char} was new in LispWorks 7.0. In LispWorks 6.1 and earlier versions \texttt{simple-char} has the most similar meaning. \texttt{bmp-char} has no obvious equivalent in LispWorks 6.1 and earlier versions, where \texttt{simple-char} is the closest thing, but in most cases when you used \texttt{simple-char} it actually better to use \texttt{cl:character} (or leave it as \texttt{simple-char}, because it is now a synonym for \texttt{cl:character}).

See also

\texttt{bmp-char-p}

\texttt{bmp-string}

\texttt{bmp-char-p} \hspace{1cm} \textit{Function}

Summary

The predicate for \texttt{bmp-char} objects.

Package

\texttt{lispworks}

Signature

\texttt{bmp-char-p object => result}

Arguments

\texttt{object} \hspace{1cm} A Lisp object.

Values

\texttt{result} \hspace{1cm} A boolean.

Description

The function \texttt{bmp-char-p} returns \texttt{t} if its argument is a character with a code less than \texttt{#x10000}, otherwise it returns \texttt{nil}. 
Compatibility note
bmp-char-p was new in LispWorks 7.0. In LispWorks 6.1 and earlier versions simple-char-p has the most similar meaning.

See also bmp-char

bmp-string

simple-bmp-string

Summary
String types that hold bmp-chars.

Package lispworks

Description
The type bmp-string is a string that can hold bmp-chars, that is characters with code below #x10000 (that is 16-bit). This corresponds to the Basic Multilingual Plane of Unicode.

simple-bmp-string is the simple version of bmp-string, that is it is a simple-array of bmp-chars.

Notes
1. bmp-strings use less memory than cl:character strings (type text-string), but cannot hold supplementary characters (that is, characters with code #x10000 or greater).

2. The corresponding character type is bmp-char.

Compatibility note
bmp-string was new in LispWorks 7.0. In LispWorks 6.1 and earlier versions text-string is similar to bmp-string. However, in most cases where you use text-string you probably still want to use text-string (using its new meaning, covering all the Unicode range).

See also text-string
bmp-char
"Character and String types" on page 436
### bmp-string-p

**Summary** The predicates for `bmp-string` and `simple-bmp-string`.

**Package** lispworks

**Signature**
- `bmp-string-p object => result`
- `simple-bmp-string-p object => result`

**Arguments**
- `object` A Lisp object.

**Values**
- `result` A boolean.

**Description** The functions `bmp-string-p` and `simple-bmp-string-p` test whether their argument is a `bmp-string` or a `simple-bmp-string` respectively.

**See also** `bmp-string` `simple-bmp-string`

### *browser-location*

**Variable**

**Signature** `*browser-location*`

**Package** lispworks

**Initial value** `nil`

**Description** Controls how the online documentation interface and the function `open-url` find a web browser executable (either Netscape, Firefox, Mozilla or Opera) to use. The value should be `nil` or a string.

If the value is `nil`, LispWorks attempts to find the browser using the value of the environment variable `PATH`.
If the value is a string, it specifies the directory in which the browser is installed. Typical values are "/usr/bin/" and "/usr/local/bin/".

**Note:** do not omit the trailing slash.

**Note:** *browser-location* is used only in the Motif-based IDE.

**See also** open-url

---

**call-next-advice**

**Summary**
Calls the next piece of advice associated with a function.

**Package**
lispworks

**Signature**
call-next-advice args

**Arguments**
args are arguments to be given to the next piece of advice to be called. Any number of arguments may be given in this way, including keyword arguments, and there is no requirement for pieces of around advice to receive the same number of arguments as the original definition expected.

**Values**
call-next-advice returns the values produced by the call to the next piece of advice (or to the combination of before and after advice and the original definition).

**Description**
call-next-advice is the local function used to invoke the next item in the ordering of pieces of advice associated with a function. It can only be called from within the scope of the around advice. Advice may be attached to a function by defadvice and this allows the behavior of a function to be modified. Extra code to be performed before or after the function may be simply added by creating before or after advice for it. Around advice is more powerful and replaces the orig-
inal definition. All the advice for a function is ordered with the around advice coming first.

The first piece of around advice receives the arguments to the function and may return any values at all. It has access to the rest of the advice, and to the original definition, by means of \texttt{call-next-advice}. A call to this from within the body of the around advice invokes the next piece of around advice with the arguments given to \texttt{call-next-advice}. The last piece of around advice in the ordering invokes the sequence of before advice, the original definition, and after advice if it calls \texttt{call-next-advice}. Around advice may contain any number of calls to \texttt{call-next-advice}, including no calls.

\section*{Notes}

1. \texttt{call-next-advice} is an extension to Common Lisp. See Chapter 6, “The Advice Facility” for a broader discussion of advice.

\texttt{call-next-advice} is not like \texttt{cl:call-next-method}, where passing no arguments has a special meaning. To pass the same arguments to the next advice, you need something like:

\begin{verbatim}
(lw:defadvice (my-func my-func-advice :around)
  (a b c &rest other-args)
  (format t "my-func advice\n")
  (apply #'lw:call-next-advice a b c other-args)
)

or

(lw:defadvice (my-func my-func-advice :around)
  (&rest args)
  (format t "my-func advice\n")
  (apply #'lw:call-next-advice args)
)
\end{verbatim}

See also \texttt{defadvice}

Chapter 6, “The Advice Facility”
choose-unicode-string-hash-function

Function

Summary
Returns a hash function suitable for strings, ignoring case using specified Unicode rules.

Package
lispworks

Signature
choose-unicode-string-hash-function &key style => hash-function

Arguments
style A keyword

Values
hash-function A hash function

Description
The function choose-unicode-string-hash-function return a function which is suitable for use as the hash-function argument to make-hash-table. The function hash-function generates a hash value for a string, ignoring case using specified Unicode comparison rules specified by style.

The current implementation only supports one value of style:
:simple-case-fold
Compares each character of the string using the simple case folding rules in Unicode 6.3.0.

See also
make-hash-table
unicode-string-equal

compile-system

Function

Summary
The function compile-system compiles all the files in a system necessary to make a consistent set of object files.

Package
lispworks
Signature

\texttt{compile-system \textit{system-name} \&key force simulate load args target-directory}

Arguments

\textit{system-name} \hspace{1em} A symbol representing the name of the system. The system must have been defined already using the \texttt{defsystem} macro.

\textit{force} \hspace{1em} If \texttt{t} then all the files in the system are compiled regardless. (This argument was formerly called \texttt{force-p}. The old name is currently still accepted for compatibility.)

\textit{simulate} \hspace{1em} If \texttt{nil} or not present then \texttt{compile-system} works silently. Otherwise a plan of the actions which \texttt{compile-system} intends to carry out is printed. What happens next depends on the value of \textit{simulate}:

\texttt{t} — do nothing.

\texttt{:ask} — you are asked if you wish the plan to be carried out using \texttt{y-or-n-p}.

\texttt{:each} — \texttt{compile-system} displays each action in the plan one at a time, and asks you whether you want to carry out this particular action. The answer \texttt{c} executes the rest of the plan without further prompting, returns from \texttt{compile-system} without further processing, and \texttt{y} and \texttt{n} work as expected.

\texttt{:simulate} may be abbreviated as \texttt{:sim}.

\textit{load} \hspace{1em} If \texttt{t} then \texttt{load-system} is called after \texttt{compile-system} has finished. If \texttt{:no} then no files are loaded at all. The default is \texttt{nil}.

\textit{args} \hspace{1em} Arguments to be passed directly to the compiler.
target-directory This must be a string representing a valid directory. It defaults to the :default-pathname option to defsystem. This is the directory where the object files created are put. If the target-directory is given then dependency information expressed in the system rules is ignored. :target-directory may be abbreviated as :t-dir.

Values compile-system returns nil.

Examples

(compile-system 'blackboard :simulate :ask)

(compile-system 'tms :load t)

(compile-system 'packages :load :no
   :target-directory "~/usr/users/386i/"
)

Notes

If load is t then load-system is called after the system has been compiled.

C source files, for example foo.c, can be included in a system (see the use of :default-type and :type in defsystem). The corresponding object file name is foon.so on Linux, FreeBSD, AIX and x86/x64 Solaris, foon.dylib on Mac OS X and foon.o on SPARC Solaris, where n is a platform-specific integer. On Windows the object file name is foon.dll.

See also concatenate-system defsystem load-system

concatenate-system

Function

Summary Produces a single, concatenated fasl from a defsystem system or systems.
Package lispworks

Signature `concatenate-system output system &key force simulate sim source-only args target-directory t-dir script-p => result`

```
system ::= system-name`
```

Arguments

- `output` The name of the required concatenated fasl.
- `system-name` The name of a system defined using `def-system`.
- `simulate` Verbosity conditions, see Description for more detail.
- `sim` Same as `simulate`.
- `force` If `t`, then all files in the system will be concatenated.
- `source-only` If `t`, the source files of the system are concatenated.
- `target-directory` The directory to search for the object files.
- `t-dir` Same as `target-directory`.

Values `result` A list containing the name or names of the concatenated systems.

Description

This function produces a single, concatenated fasl, `output-file`, from a list of individual systems (named amongst the `args`).

Since concatenated fasl files may be produced in this way, you do not need to be wary of MS filename conventions if developing sources on UNIX for a Microsoft Windows application. This clearly allows more freedom for naming source files. However, `output-file` must, in such cases, be a MS-Windows-compatible filename.

If `simulate` is `nil` or is not present, `concatenate-system` will work silently. Otherwise, a plan of the actions which
concatenate-system intends to carry out is printed. What happens next depends upon the value of simulate:

- If it is t, the function does nothing.
- If :ask, then the user is asked, using y-or-n-p, if the plan should be carried out.
- If it is :each, the user is asked at each stage in the plan if the current action should be carried out. The responses y and n work as normal. If e is typed, concatenate-system exits without further processing.

If source-only is t, files will be loaded only if they are sources.

If, when searching target-directory for an object file, the file cannot be found, the appropriate source file from the system’s default directory will be loaded instead.

See also

- compile-system
- defsystem
- load-system

### copy-file

**Function**

**Summary**
Copies the contents of a file to another file.

**Package**
lispworks

**Signature**
copy-file from to

**Arguments**

- from A pathname designator.
- to A pathname designator.

**Description**
The function copy-file copies the contents of the file from to another file. The file from must exist.
copy-file opens from for input and to for output using if-exists :supercede (see cl:open in the Common Lisp Hyper-Spec) and copies the contents from from to to.

On any failure copy-file signals an error.

copy-file does not return a useful value.

See also append-file

count-regexp-occurrences

Function

Summary
Count the occurrences of a pattern in a string.

Package
lispworks

Signature
count-regexp-occurrences pattern string &key start end overlap case-sensitive  =>  count

Arguments
pattern  A string or precompiled regular expression object.
string  A string.
start, end  Bounding index designators of string.
overlap  A generalized boolean.
case-sensitive  A generalized boolean.

Values
count  An integer.

Description
The function count-regexp-occurrences counts the occurrences of pattern in the part of string bounded by start and end.

If pattern is a string, count-regexp-occurrences precompiles it first. If you use count-regexp-occurrences with the same pattern string several times, it is better to precompile it using precompile-regexp.
The LISPWORKS Package

start and end have the same meaning as in count and other Common Lisp sequence functions.

case-sensitive controls whether a string pattern is precompiled as a case sensitive or case insensitive search. A non-nil value means a case sensitive search. The value nil (the default) means a case insensitive search.

If overlap is false (the default), then count-regexp-occurrences counts matches that to not overlap. If overlap is non-nil, matches can overlap, and count-regexp-occurrences finds all of the ways in which the pattern can be matched inside string.

The regular expression syntax used by count-regexp-occurrences is similar to that used by Emacs, as described in the "Regular expression syntax" section of the LispWorks Editor User Guide. If you use Help > Search to locate this section in the LispWorks IDE, then select the Contents radio button.

Examples

(count-regexp-occurrences "aaa" "aaaaa")
=> 1

(count-regexp-occurrences "aaa" "aaaaa" :overlap t)
=> 3

(count-regexp-occurrences "12" "81267124")
=> 2

(count-regexp-occurrences "12" "81267124" :start 4)
=> 1

(let* ((path (example-file "capi/elements/text-input-pane.lisp"))
   (file-string (file-string path)))
   (count-regexp-occurrences ":title" file-string))
=> 20 ; in LispWorks 7.1
See also  
find-regexp-in-string  
precompile-regexp

**current-pathname**  
*Function*

**Summary**
Computes a pathname relative to the current path.

**Package**
lispworks

**Signature**
current-pathname &optional relative-pathname type => pathname

**Arguments**
relative-pathname  A pathname designator.

  type  A string or nil.

**Values**
pathname  A pathname.

**Description**
The function `current-pathname` is useful for loading other files relative to a file.

`current-pathname` computes a pathname from the current operation as follows:

When loading a file
- Uses `*load-pathname*`.

When compiling a file
- Uses `*compile-file-pathname*`.

When evaluating or compiling an Editor buffer
- Uses the pathname of the buffer, if available, otherwise uses the current working directory.

Otherwise
- Uses the current working directory.
The pathname computed above is then translated to a physical pathname, and the argument relative-pathname is merged with this physical pathname. The pathname-type of the result pathname is set to type if supplied, the pathname-version is set to :newest, and pathname is returned.

A useful value for type is nil, which can be used to allow load to choose between lisp or fasl regardless of the type of the current pathname.

Notes
defsystem uses current-pathname with its :default-host argument.

Examples
Suppose you want the file foo to load the file bar.

While loading the source file foo.lisp:

```lisp
(current-pathname "bar")
=> 
#P"C:/temp/bar.lisp"
```

While loading the binary file foo.ofasl:

```lisp
(current-pathname "bar")
=> 
#P"C:/temp/bar.ofasl"
```

To load bar.lisp or bar.ofasl according to the value of *load-fasl-or-lisp-file*, regardless of whether foo.lisp or foo.ofasl is being loaded, specify type nil:

```lisp
(load (current-pathname "bar" nil))
```

See also
defsystem
pathname-location

defadvice

Macro

Summary
Defines a new piece of advice.

Package
lispworks
Signature

\[
\text{defadvice (function-dspec name advice-type &key where documentation)}
\]

\[
\text{lambda-list \&body body } \Rightarrow \text{nil}
\]

\[
\text{advice-type ::= \text{:before | :after | :around}}
\]

Arguments

\text{function-dspec} \quad \text{A function-dspec Specifies the function definition to which the piece of advice belongs. See “Function dspecs” on page 84 for description of function-dspec.}

\text{name} \quad \text{A symbol naming the piece of advice being created. It should of course be unique to the advised function, but does not need to be globally unique.}

\text{advice-type} \quad \text{A keyword specifying the kind of advice wanted.}

\text{where} \quad \text{Specifies where this advice should be placed in the ordering of pieces of advice for the function. By default a piece of advice is placed at the start of the corresponding section. If this argument is present and is \text{:end} then the advice is instead placed at the end of its section. The other permissible value for this argument is \text{:start}, which places the advice at the start of its section in the ordering (as in the default behavior).}

\text{documentation} \quad \text{A string providing documentation on the piece of advice.}

\text{lambda-list} \quad \text{A lambda list for the piece of advice. In the case of before and after advice this should be compatible with the lambda list for the original definition, since such advice receives the same arguments as that function.}

\text{body} \quad \text{The main body of the advice.}

Values

\text{defadvice returns nil.}
Description

defadvice is the macro used to define a new piece of advice. Advice provides a way to change the behavior of existing functional definitions in the system. In a simple instance advice might be used to carry out some additional actions before or after the original definition. More sophisticated uses allow the definition to be replaced by new code that can access the original function repeatedly or as rarely as desired, and that can receive different numbers of arguments and return any values. A function may have any number of pieces of advice attached to it by using defadvice.

When function-dspec names a macro, then the function with which the advice is associated is the expansion function for that macro. Thus before and after advice for a macro receive the arguments given to the macro expansion function, which are normally the macro call form and an environment.

There are three kinds of advice that may be defined: before, after and around advice. The first two kinds attach auxiliary code to be carried out alongside the original definition (before it for before advice, after it in the case of after advice). Around advice replaces the function altogether; it may define code that never accesses the original definition, that receives different numbers of arguments, and returns different values. All the pieces of advice for a function are ordered. The ordering is important in determining how all the pieces of advice for a function are combined. Around advice always comes first, then before advice, then the original definition, and lastly the after advice.

Conceptually the before advice, the original definition and the after advice are amalgamated into one new construct. If this gets called then each of its components receives the same arguments in turn, and the values returned are those produced by the last piece of after advice to be called in this way (or the original function if there is no after advice). The code associated with before and after advice should not destructively modify its arguments.
If around advice is present then the first piece of around advice is called, instead of the combination involving before and after advice discussed above. It does not have to access any of the other advice, nor the original definition. Its only link to the rest of the advice is by means of a call to call-next-advice. It may invoke this as often as it chooses, and by doing so it accesses the next piece of around advice if present, or else it accesses the combination of before and after advice together with the original definition.

Remove advice using remove-advice or delete-advice.

Notes
defadvice is an extension to Common Lisp.

See also
call-next-advice
delete-advice
remove-advice
Chapter 6, “The Advice Facility”

*default-action-list-sort-time*

Variable

Summary Determines when actions in action lists are sorted.

Package lispworks

Signature *default-action-list-sort-time*

Initial value :execute

Description The variable *default-action-list-sort-time* is a keyword that is either :execute or :define-action, denoting when actions in action-lists are sorted (see define-action-list for an explanation of ordering specifiers). Actions are sorted either at time of definition (:define-action) or when their action-list is executed (:execute). The default sort time is :execute.
See also  
define-action
define-action-list

*default-character-element-type*  

Parameter

Summary  Provides defaults for all character type parameters.

Package  lispworks

Description  This parameter provides defaults for all character type parameters. The legal values are cl:base-char, lw:bmp-char and cl:character. lw:simple-char is also supported for backwards compatibility.

Its value must only be set via a call to set-default-character-element-type.

This is intended for efficiency of applications with only 8-bit strings, where you can do

(set-default-character-element-type 'base-char)

and also for efficiency of applications with only 16-bit strings, where you can do

(set-default-character-element-type 'lw:bmp-char)

If your program uses 16-bit or 32-bit strings you should already be aware of these issues, and make some attempt to provide explicit types.

When the compiler does type inferencing it behaves as if this variable was bound to cl:character; if you want assumptions about types to be hard-coded into your program, you must supply explicit declarations and type arguments.

See also  
string
simple-string
make-string
open
define-action  

**Summary**

 Adds a new action to a specified list.

**Package**

 lispworks

**Signature**

 define-action name-or-list action-name data &rest specs =>

**Arguments**

 name-or-list A list or action list object.

 action-name A general lisp object.

 data An object.

 specs A list.

**Description**

 The macro `define-action` adds a new action to the action list specified by `name-or-list`; this action will be executed according to the action-list's execution-function (see `execute-actions`) when executed. If the action-list specified by `name-or-list` does not exist, then this is handled according to the value of `*handle-missing-action-list*`.

 `name-or-list` is evaluated to give either a list UID (to be looked up in the global registry of lists) or an action list object. `action-name` is a UID (general lisp object, to be compared by `equalp`). It uniquely identifies this action within its list (as opposed to among all lists).

 `data` specifies an object referring to data relevant to the action.

 `specs` is a free-form list of ordering specifiers and extra keywords, used to control more details of how and when this action is executed.

 Action-items are normally expected not to be redefined. If an action-item with that action-name already exists in the
action-list (that is, one with an identifier `equalp` to the action-name), then the notification and subsequent handling of this attempt is controlled by the values in the list `*handle-existing-action-in-action-list*`. This is to prevent problems due to re-evaluating an action definition inappropriately. Notification and redefine behavior can be overridden by using the `:force` keyword argument. In this case, any required redefinition is performed unconditionally and without notification.

The following keywords are recognized in the `specs` argument:

- **:after**
  - The following element in `specs` is a UID.
  - `:after` specifies that the action-item being defined must be run after the action-item named. If there is no action-item with a matching name, the restriction is ignored.

- **:before**
  - Like `:after`, but this action-item must be run before the one specified.

- `:after` and `:before` can be specified as many times as necessary to describe the ordering constraints of this action-item with respect to its neighbors.

- **:once**
  - Specifies that this action-item should be executed only once; after execution, it is disabled.

- **:force**
  - Specifies that this definition should override any previous definition of this action-item, rather than be subject to the value of `*handle-existing-action-in-action-list*`.

Example

```
(define-action :network-startup "Reset decnet buffers"
  '(decnet::reset-network-buffers
    *net-buffers*)
  :after "Reset core network"
  :once))
```
define-action-list  

**Summary**  
Defines a registered action list.

**Package**  
lispworks

**Signature**  
define-action-list uid &key documentation sort-time dummy-actions default-order execution-function =>

**Arguments**  
- **uid**  
  A Lisp object.
- **documentation**  
  A string.
- **sort-time**  
  One of :execute or :define-action.
- **dummy-actions**  
  A list.
- **default-order**  
  A list.
- **execution-function**  
  A function.

**Description**  
The macro **define-action-list** defines an action list.  
uid is a unique identifier, and must be a general Lisp object, to be compared by equalp. It names the list in the global registry of action lists. See make-unregistered-action-list to create unnamed, “unregistered” action-lists. The uid may be quoted, but is not required to be. It is possible, but not recommended, to define an action-list with unique identifier nil. If a registered action-list with the uid already exists (that is, one which returns t when compared with equalp), then notification and subsequent handling is controlled by the value of the variable *handle-existing-action-list*.

The documentation string allows you to provide documentation for the action list.
sort-time is a keyword specifying when added actions are sorted for the given list — either :execute or :define-action (see *default-action-list-sort-time*).

dummy-actions is a list of action-names that specify placeholder actions; they cannot be executed and are constrained to the order specified in this list, for example

' (:beginning :middle :end)

default-order specifies default ordering constraints for subsequently defined action-items where no explicit ordering constraints are specified. An example is

' (:after :beginning :before :end)

execution-function specifies a function that you define. It must accept arguments of the form:

( the-action-list other-args-list &rest keyword-value-pairs )

where the two required arguments are the action-list and a list of additional arguments passed to execute-actions, respectively. The remaining arguments are any number of keyword-value pairs that may be specified in the call to execute-actions. If no execution function is specified, then the default execution function will be used to execute the action-list.

See the manual entries for with-action-list-mapping and with-action-item-error-handling for examples of execution-functions.

To add an action to an action list you have defined, use define-action.

See also

*default-action-list-sort-time*  
define-action  
*handle-existing-action-list*  
undefine-action-list  
with-action-item-error-handling  
with-action-list-mapping
defsystem

**Macro**

**Summary**

`defsystem` is used to define systems for use with the LispWorks system tools. A system is a collection of files and other systems that, together with rules expressing the interdependencies of those files and subsystems, make a complete program. The LispWorks system tools support the development and maintenance of large programs. Find a full description at “Common Defsystem and ASDF” on page 305.

**Package**

`lispworks`

**Signature**

`defsystem name options &key members rules => system-name`

**Arguments**

- `name` A string or a symbol, not evaluated.
- `options` A list of keyword-value pairs.
- `members` A list of strings or lists.
- `rules` A list.

**Values**

- `system-name` A string.

**Description**

The name of the system to be made is a string specified by `name`. If `name` is a symbol, then its symbol name is used.

`options` are expressed as a list of keyword-value pairs. The following keywords are recognized:

- `:package` The default package that files are compiled and loaded in. If not specified, this defaults to the value of `package*` at macroexpansion time.
:default-pathname

Used to compute a default pathname in which to find files. defsystem uses current-pathname to compute the pathname. defsystem checks that all the files given as members actually exist.

:default-host
The root pathname of a system is defined to be the :default-host if it is given. Otherwise, it is taken to be the directory containing the defsystem file.

Absolute pathnames are interpreted literally, and relative pathnames are taken relative to the root pathname.

:default-type
This is the default type of the members of the system. This may be :lisp-file, :lsp-file, :c-file, or :system.


The com module adds the type :midl-file and the automation module adds :midl-type-library-file.

The default is :lisp-file, which means files with file type (extension) "lisp".

:documentation
This is a string.

:object-pathname

A string or pathname specifying a directory where object files are written.

Note: This option will not work if the names in members represent absolute pathnames.
:optimize A declaration specifying default compilation qualities within the scope of compile-system. These settings override the current global setting. They can be overridden per member by the :optimize option (for subsystems) or proclaim (in files). The :optimize defsystem option accepts the same optimize qualities as proclaim and which are fully described in “Compiler control” on page 104. See below for examples.

members is a list defining the members of the system. Elements of the list may be a string name representing the name of the physical file or system referred to. Elements of the list may also be a symbol, which is interpreted as its symbol name.

Elements of the members list can also be a list of the form (name {keyword value}*) where name is once again a string or a symbol naming a file or system.

The members of the system name must have unique names, as compared by equalp. For example, if members contains "foo" then there cannot be another member (either a file or a system) named "foo", "FOO" or foo.

The possible keywords and their values are:

:type The type of this member. Allowed values are as for :default-type. If not specified it defaults to the value of :default-type given as an option.

:root-module If nil then this member is not loaded unless its loading is specifically requested as a result of a dependency on another module

:source-only Only the source file for this member is ever loaded
The LISPWORKS Package

:load-only  The member is never compiled by defsystem, objects are loaded in preference to source files

:load-for-compile-only  The member is only loaded as necessary during compilation and is never loaded independently

:features  The member is only considered during planning if the feature expression is true.

:package  A default package for the member.

:embedded-module  Only allowed when the value for :type is :c-file. The value embedded-module is used to create a FLI embedded module named embedded-module instead of loading the object file. See fli:install-embedded-module in the LispWorks Foreign Language Interface User Guide and Reference Manual for how to load the embedded module.

On Windows, the automation module adds the keyword :com for a member with type :midl-type-library-file. Then a member of the form

("ms97.tlb" :type :midl-type-library-file :com nil)

can be specified when you use only Automation client code, reducing the memory used.

rules is a list of rules of the following format:

{{{in-order-to} action {all | {{ member-name }}* }}
   (:caused-by {{action {previous |{{member-name }}* }}}*})
   (:requires {{action {previous |{{ member-name }}*}})*})

The keyword :all refers to all the members of the system. It provides a shorthand for specifying that a rule should apply to all the system’s members. The keyword :previous refers
to all the members of the system that are before the member in the list of members. This makes it easy, for example, to specify that in order to compile a file in a system, all the members that come before it must be loaded.

The name of the system is returned.

There are more details about the rules in “DEFSYSTEM rules” on page 308.

Examples

(defsystem defsys-macros
 (:default-pathname "/usr/users/james/scm/defsys/
 :default-type :lisp-file
 :package defsystem)
 :members ("new-macros"
 "scm-timemacros")

(defsystem clos-sys
 (:default-pathname "/usr/users/clc/defsys/
 :default-type :lsp-file
 :package defsystem)
 :members
 ("defsys-macros" :type :system :root-module nil)
 "class"
 "time-methods"
 ("scm-pathname" :source-only t)
 "execute-plan"
 "file-types"
 "make-system"
 "conv-defsys")
 :rules
 ((:in-order-to :compile ("class" "time-methods")
  (:caused-by (:compile *defsys-macros*))
  (:requires (:load "defsys-macros")))
 (:in-order-to :compile
  ("time-methods" "execute-plan")
  (:requires (:load "class")))))

1113
(defsystem dataworks-demo
 (:default-type :system)
 :members (  
 "db-class"
 "planar"
 "dataworks-dep"
 "dataworks-interface-tk"
 "dataworks-interface-tools"
 "drugs-demo"
 ("gen-demo" :type :lisp-file)
 ("load-icon" :type :lisp-file :source-only t)
 )
 :rules ((:in-order-to :compile :all
 (:requires (:load :previous))))

This example illustrates the use of :optimize.

(defsystem foo (:optimize ((speed 3) (space 3)
 (safety 0)))
 :members ("bar"
 "baz")
 :rules ((:compile :all
 (:requires (:load :previous))))

This last example illustrates the use of :embedded-module.

(defsystem my-foreign-code ()
 :members
 ("my-c-code.c" :type :c-file
 :embedded-module my-module))

Then initialize at run time with

(fli:install-embedded-module 'my-module)

Notes

1. Subsystems must be defined before any system of which they are part.

2. The order of members is important and reflects the order in which operations are carried out on the members of the system, subject to the rules.

See also

load-system
compile-system
concatenate-system
current-pathname
*defsystem-verbose*

*defsystem-verbose*  
**Variable**  
Summary Controls the amount of messages printed by defsystem about system (re)definition.  
Package lispworks  
Initial value  
Description The variable *defsystem-verbose* is a generalized boolean controlling the amount of messages printed by defsystem.  
When the value is true, the system prints messages about system definition and redefinition. The default value is t.  
See also defsystem

delete-directory  
**Function**  
Summary Deletes a directory.  
Package lispworks  
Signature delete-directory directory &optional error => result  
Arguments directory A pathname designator.  
error nil, :error or :no-error.  
Value result t or nil.
Description

The function `delete-directory` attempts to delete the directory `directory`. It returns `t` on success, and on failure either returns `nil` or signals an error.

`error` determines what happens when `delete-directory` fails. When `error` is `nil` (the default), if `directory` does not exist `delete-directory` returns `nil`, otherwise any failure causes an error to be signaled. If `error` is `:no-error`, `delete-directory` returns `nil` on any failure. If `error` is `:error`, any failure causes an error to be signaled.

Typical reasons for failures in `delete-directory` are that `directory` is not empty, or that the user does not have the right permissions.

---

### deliver

**Function**

**Summary**
The main interface to the Delivery tools.

**Package**
lispworks

**Signature**
deliver function file level &rest keywords

**Description**
The function `deliver` is the main interface to the LispWorks delivery tools. You use it to create LispWorks executable applications and dynamic libraries.

For more information about Delivery including a detailed description of `deliver`, see the *LispWorks Delivery User Guide*.

For information about invoking `deliver` using the IDE, see "The Application Builder" in the *LispWorks IDÉ User Guide*.

**See also**
delivered-image-p
save-image
"Guidance for control of the memory management system” on page 130
*describe-length*  
**Variable**

**Summary**  
Determines how many attributes of a composite object are described.

**Package**  
lispworks

**Initial value**  
20

**Description**  
The variable *describe-length* controls how many attributes of a composite object the function describe describes.

This means the number of elements of a sequence, entries in a hash table, slots of a structure instance, and so on.

If *describe-length* is nil then describe describes all of the attributes. Use this value only with care.

**Notes**  
The describe functionality is load-on-demand in the LispWorks image as shipped. Therefore if you have not done (require "describe") or called describe, *describe-length* may be unbound.

**See also**  
describe

*describe-level*  
**Variable**

**Summary**  
Controls the depth to which describe describes arrays, structures and conses.

**Package**  
lispworks

**Initial value**  
1
**Description**

The variable `*describe-level*` controls the depth to which the function `describe` describes arrays, structures and conses.

**Notes**

The `describe` functionality is load-on-demand in the LispWorks image as shipped. Therefore if you have not do `(require "describe")` or called `describe, *describe-level*` may be unbound.

**Example**

```
CL-USER 23 > (describe 1)
[... load output not shown ...]

1 is a BIT
DECIMAL      1
HEX          1
OCTAL        1
BINARY       1

CL-USER 24 > *describe-level*
1

CL-USER 25 > (defstruct foo a s d)
FOO

CL-USER 26 > (defmethod describe-object ((f foo) (s stream))
    (format s "FOO ~S~%" f)
    (describe (foo-a f) s))
#<STANDARD-METHOD DESCRIBE-OBJECT NIL (FOO STREAM) 2068295C>

CL-USER 27 > (describe (make-foo :a (vector 1 2 3) :s 42))

FOO #S(FOO A #(1 2 3) S 42 D NIL)
#(1 2 3)

To make `describe` operate on objects inside the structure instance, increase the value of `*describe-level*`:
```
CL-USER 28 > (setf *describe-level* 2)
   2

CL-USER 29 > (describe (make-foo :a (vector 1 2 3) :s 42))

FOO #S(FOO A #(1 2 3) S 42 D NIL)
#(1 2 3) is a SIMPLE-VECTOR
  0   1
  1   2
  2   3

See also   describe

*describe-print-length*  
Variable

Summary   Specifies a print length for describe and apropos.

Package   lispworks

Initial value   10

Description   If *print-length* is nil, describe and apropos bind *print-length* to the value of *describe-print-length*.

See also   describe

*describe-print-level*  
Variable

Summary   Specifies a print level for describe and apropos.

Package   lispworks

Initial value   10
The LISPWORKS Package

Description
If *print-level* is nil, describe and apropos bind *print-level* to the value of *describe-print-level*.

See also
describe

dll-quit

Function

Summary
Makes a LispWorks dynamic library quit.

Package
lispworks

Signature
dll-quit &key kill-all-processes timeout output force => result, quit-output

Arguments
kill-all-processes A generalized boolean.
timeout A positive integer or nil.
output An output stream designator.
force A generalized boolean.

Values
result t or nil.
quit-output A string or nil.

Description
The function dll-quit makes a LispWorks dynamic library (or DLL) quit on returning from the callback in which it was called. It must be called only:

• In an image running as a dynamic library, meaning an image created by save-image with :dll-exports or by deliver with :dll-exports, and

• Inside the dynamic scope of a callback into the dynamic library. That is, not in a process that was started by process-run-function.

dll-quit sets up the internal state such that just before returning into its caller in the LispWorks dynamic library it
causes LispWorks to quit. After quitting the callback returns as normal. The library can be unloaded using FreeLibrary, or you can re-use it (without re-loading).

By default kill-all-processes is nil which means that, if there are other running processes, dll-quit just returns nil. If kill-all-processes is non-nil, dll-quit tries to kill all the other processes, and if it succeeds, it quits.

If kill-all-processes is true, timeout is a maximum time to wait after killing the other processes. It allows timeout seconds for all processes to die.

dll-quit should be called when no other processes are running, whether they were created by a callback or by process-run-function. If such processes exist, by default dll-quit does nothing and returns nil. If force is non-nil, dll-quit always tries to set LispWorks up for quitting. LispWorks will quit even after a failure to kill all other processes and complete any required shut down operations. A true value of force automatically implies kill-all-processes true.

However, if any of the other processes is stuck in a foreign call, the quitting may fail to finish properly. The default value of force is nil.

If output is supplied, dll-quit generates output if it is called when other processes are still running, or a required shut down operation was not completed. output can be an output stream, t (interpreted as *standard-output*) or nil. If output is nil, dll-quit collects the output and returns it as second argument quit-output. Otherwise it writes the output to the stream and quit-output is nil.

The output contains a list of the other processes that are still running. If kill-all-processes or force was supplied, and killing the other processes failed, the output also contains backtraces of the other processes, and possibly other debugging information.
result is t on success: the LispWorks dynamic library is set to quit on returning from the callback. result is nil when other processes are running: the image is not set to quit.

quit-output contains the output which was generated when output nil was passed. Otherwise quit-output is nil.

If dll-quit is called inside a recursive foreign callback, the LispWorks dynamic library quits only when the outermost callback returns.

Notes

1. dll-quit is intended for use when a LispWorks dynamic library is loaded by a main process which you (the LispWorks programmer) do not control. If you control the main process, then use QuitLispWorks instead.
   It is expected that the main process will call into the dynamic library with some "shutdown" call, and then calls FreeLibrary to free the library. The shutdown call should close and free everything that needs to be closed or freed, call dll-quit, and return.

2. dll-quit is supported only where LispWorks can be a dynamic library. Currently this is in LispWorks on Microsoft Windows, Intel Macintosh, Linux, x86/x64 Solaris and FreeBSD.

See also
deliver
save-image

dotted-list-length

Function

Summary
Similar to list-length

Package
lispworks

Signature
dotted-list-length list => result
Arguments  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>A list.</td>
</tr>
</tbody>
</table>

Value  

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>An integer.</td>
</tr>
</tbody>
</table>

Description  

The function **dotted-list-length** performs the same action as **list-length**, except that if the last **cdr** is not **nil** then instead of signaling an error, it returns the number of **conses** plus 1.

See also  

**dotted-list-p**

---

**dotted-list-p**  

**Function**

Summary  

Tests whether a **cons** is a list ending in a non-nil **cdr**.

Package  

**lispworks**

Signature  

**dotted-list-p**  

list => result

Arguments  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>A list, which must be a <strong>cons</strong>.</td>
</tr>
</tbody>
</table>

Values  

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>A generalized boolean.</td>
</tr>
</tbody>
</table>

Description  

The function **dotted-list-p** is a predicate which tests whether **list** (which must be a **cons**) is a list ending in a non-nil **cdr**. It returns true if this is the case, otherwise it returns false.

See also  

**dotted-list-length**

---

**do-nothing**  

**Function**

Summary  

Ignores its arguments and returns an unspecified value.

Package  

**lispworks**
Signature

**do-nothing** &rest **ignore** => **unspecified**

Arguments

**ignore**

All arguments are ignored.

Values

**unspecified**

An unspecified value.

Description

The function **do-nothing** ignores its arguments and returns an unspecified value. It is useful as a function argument.

See also

**false**

**true**

*enter-debugger-directly*  

Variable

Summary

Controls direct entry into the Debugger tool.

Package

lispworks

Initial value

nil

Description

The variable *enter-debugger-directly* is a generalized boolean which affects the behavior of the LispWorks IDE when an error is signaled outside of the Listener REPL.

Value nil causes an error notifier window to be displayed (from which you can abort, report a bug, or raise a Debugger tool).

A true value causes the Debugger tool to be displayed immediately, and no error notifier appears.

Notes

Errors signaled in a Listener Read-Eval-Print loop are handled in the REPL and therefore *enter-debugger-directly* has no effect on the behavior in this case.
**environment-variable**  

**Function**

**Summary**
Reads the value of an environment variable from the environment table of the calling process.

**Package**
lispworks

**Signature**

\[
\text{environment-variable \hspace{5pt} name} \Rightarrow \text{result}
\]

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A string.</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>A string or nil.</td>
</tr>
</tbody>
</table>

**Description**
The function `environment-variable` reads the environment variable specified by `name` and returns its value, or `nil` if the variable could not be found.

A `setf` method is also defined, allowing you to set the value of an environment variable:

\[
\text{(setf \hspace{5pt} (environment-variable \hspace{5pt} name) \hspace{5pt} value)}
\]

If `value` is a string, then `name` is set to be `value`. If `value` is `nil` then `name` is removed from the environment table.

**Example**

In this first example the value of the environment variable `PATH` is returned:

\[
\text{(environment-variable \hspace{5pt} "PATH")}
\]

The result is a string of all the defined paths:

```
"c:\hqbin\nt\x86;c:\hqbin\nt\x86\perl;c:\hqbin\win32;c:\usr\local\bin;c:\WINNT35\system32;c:\WINNT35;C:\MSTOOLS\bin;c:\TGS3D\PROGRAM;c:\program files\devstudio\sharedide\bin\ide;c:\program files\devstudio\sharedide\bin;c:\program files\devstudio\vc\bin;c:\msdev\bin;c:\WINDOWS;C:\WINDOWS\COMMAND;C:\WIN95\COMMAND;C:\MSINPUT\MOUSE"
```
In the second example, the variable `MYTZONE` is found not to be in the environment table:

```
(environment-variable "MYTZONE")
NIL
```

It is set to be GMT using the `setf` method:

```
(setf (environment-variable "MYTZONE") "GMT")
```

See also “Accessing environment variables” on page 458

---

**errno-value**

*Function*

**Summary**

Returns the current value of the UNIX variable `errno`.

**Package**

`lispworks`

**Signature**

`errno-value => value`

**Arguments**

None.

**Values**

`value` The current value of `errno`.

**Description**

The function `errno-value` returns the current value of the UNIX variable `errno`.

**Notes**

`errno-value` is implemented only on non-Windows platforms.

**Example**

```
USER 10 > (errno-value)
2
USER 11 > (get-unix-error 2)
"no such file or directory"
```

See also `get-unix-error`
**example-file**  
*Function*  

**Summary**  
Returns a path in the `examples` folder.

**Package**  
lispworks

**Signature**  
`example-file file => path`

**Arguments**  
`file`  
A pathname designator.

**Values**  
`path`  
A pathname.

**Description**  
The function `example-file` returns an absolute path to a file `file` in the `examples` folder of the LispWorks library. It does not actually test for the existence of the file.

**Example**  
```
(example-file "capi/applications/othello.lisp") =>
#P"C:/Program Files/LispWorks/lib/7-1-0-0/examples/capi/applications/othello.lisp"
```

**See also**  
`example-compile-file`  
`example-edit-file`

---

**example-compile-file**  
*Function*  

**Summary**  
Compiles a file in the `examples` folder to a temporary output file.

**Package**  
lispworks

**Signature**  
`example-compile-file file &rest args => output-truename, warnings-p, failure-p`

**Arguments**  
`file`  
A pathname designator.

`args`  
Arguments passed to `compile-file`.  

1127
The LISPWORKS Package

Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>output-truename</td>
<td>A pathname or nil.</td>
</tr>
<tr>
<td>warnings-p</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>failure-p</td>
<td>A generalized boolean.</td>
</tr>
</tbody>
</table>

Description

The function `example-compile-file` constructs the path to `file` in the `examples` folder of the LispWorks library, and a path to an output file in a temporary location which is likely to be writable.

It then calls `compile-file` with these paths as the `input-file` and `output-file`, also passing the other `args`, and returns the values returned by `compile-file`.

See also

- `get-temp-directory`
- `example-file`
- `example-edit-file`

`example-edit-file`  

Function

Summary

Displays a file from the examples folder in an Editor.

Package

`lispworks`

Signature

`example-edit-file file`

Arguments

`file`  
A pathname designator.

Description

The function `example-edit-file` constructs the path to the file `file` in the examples folder of the LispWorks library, adding the "lisp" extension if no extension is specified, and opens the file in an Editor tool in the LispWorks IDE.

If `file` is a directory name (ending in a slash), then the list of files with "lisp" extension in that directory is displayed in the Editor.
Examples

This form displays the file lib/7-1-0-0/examples/capi/applications/othello.lisp from the Lisp-Works library:

(example-edit-file "capi/applications/othello")

See also

example-file
example-compile-file

example-load-binary-file

Function

Summary

Loads a fasl file compiled by example-compile-file.

Package

lispworks

Signature

example-load-binary-file file => result

Arguments

file A pathname designator.

Values

result A generalized boolean.

Description

The function example-load-binary-file constructs the path to an output file in a temporary location which would be used as the output-file by example-compile-file.

It then calls load on that path, and returns the value result returned by load.

See also

example-compile-file

execute-actions

Macro

Summary

Executes in sequence the actions on a given list.

Package

lispworks
**Signature**

\[
\text{execute-actions} \ \text{name-or-list} \ \&\text{rest} \ \text{keyword-value-pairs} \ \&\text{rest} \ \text{other-args} =>
\]

**Arguments**

- **name-or-list**  
  An action list
- **keyword-value-pairs**  
  See description.
- **other-args**  
  A list.

**Description**

The macro `execute-actions` executes, in sequence, the actions on the specified list. If the action-list specified by `name-or-list` does not exist, then this is handled according to the value of `*handle-missing-action-list*`. Note that `name-or-list` is evaluated.

If a user-defined execution function was specified when the action list was defined, then it should accept the following arguments:

\[
\text{(action-list} \ \text{other-args} \ \&\text{rest} \ \text{keyword-value-pairs})
\]

Note that `other-args` is passed as a single list.

If a user-defined execution function was not specified when the action list was defined, then the following default mapping occurs. Each action's data is invoked via `apply` on `other-args`:

\[
\text{(apply} \ \text{data} \ \text{other-args})
\]

This behavior is modified by the keyword-value-pairs, thus:

- If the keyword parameter `:ignore-errors-p` is non-nil, any otherwise-unhandled errors signaled inside the execution of that function will be trapped, and a warning issued. Execution continues with the next action-item. If `:ignore-errors-p` is `nil` (or not specified), then the error is not trapped.
If the keyword parameter :post-process is non-nil, the first value returned by each action is handled, according to :post-process, thus:

- :collect: collect values into list
- :and: return t only if all values are t. Return nil immediately if any value is nil
- :or: return first non-nil value

See also
- define-action
- with-action-list-mapping

---

**extended-char**

*Type*

**Summary**
The extended character type.

**Package**
lispworks

**Signature**
extended-char

**Description**
The type of extended characters. A synonym for `extended-character`, but with standard spelling.

**extended-character**

*Type*

**Summary**
The extended character type.

**Package**
lispworks

**Signature**
extended-character

**Description**
The type of extended characters.
**extended-character-p**  
*Function*

**Summary**
A predicate for extended characters.

**Package**
lispworks

**Signature**
extended-character-p object => result

**Arguments**
object The object to be tested.

**Values**
result A boolean.

**Description**
The function `extended-character-p` is a predicate for extended characters.  
`result` is `t` if `object` is an extended character, and `nil` otherwise.

**See also**
extended-character

---

**extended-char-p**  
*Function*

**Summary**
A predicate for extended characters.

**Package**
lispworks

**Signature**
extended-char-p object => result

**Arguments**
object The object to be tested.

**Values**
result A boolean.

**Description**
The function `extended-char-p` is a predicate for extended characters, only with standard spelling.  
`result` is `t` if `object` is an extended character, and `nil` otherwise.
See also  
extended-char
extended-character-p

*external-formats*

Variable

Summary
A list of the names of the defined external formats.

Package
lispworks

Initial value
Microsoft Windows platforms:

(WIN32:CODE-PAGE FLI::UNICODE-WCHAR FLI::LATIN-1-WCHAR
FLI::ASCII-WCHAR :KOI8-R :MACOS-ROMAN :UTF-32 :UTF-32BE
:UTF-16LE :UTF-16-REVERSED :UTF-16-NATIVE :UTF-8
:GBK :WINDOWS-CP936 EXTERNAL-FORMAT:DOUBLE-BYTE-TABLE-LOOKUP
:UNICODE :LATIN-1-SAFE :LATIN-1-CHECKED :LATIN-1 :EUC
:SHIFT-JIS :NIHONGO-MS :NIHONGO-EUC :NIHONGO-JIS
CHARACTER :BMP-REVERSED :BMP-NATIVE EXTERNAL-FORMAT::RAW-BASE-CHARACTER
:ASCII-TERMINAL :ASCII)

On all other platforms:

(FLI::UNICODE-WCHAR FLI::LATIN-1-WCHAR FLI::ASCII-WCHAR
:UTF-16-REVERSED :UTF-16-NATIVE :UTF-8 :GBK :WINDOWS-CP936
EXTERNAL-FORMAT:DOUBLE-BYTE-TABLE-LOOKUP :JIS
:LATIN-1-CHECKED :LATIN-1 :EUC :SHIFT-JIS
:NIHONGO-MS :NIHONGO-EUC :NIHONGO-JIS EXTERNAL-FORMAT::HOST-PORTABLE
EXTERNAL-FORMAT::LATIN-PORTABLE
CHARACTER :BMP-REVERSED :BMP-NATIVE EXTERNAL-FORMAT::RAW-BASE-CHARACTER
:ASCII-TERMINAL :ASCII)

Description
The variable *external-formats* contains a list of the names of the defined external formats.

The platform-specific external format names are:

code-page
Uses the encoding in the Microsoft Windows code page specified by the :id parameter.

**latin-portable**

Intended for use when communicating with X servers, for example when passing XLFD names. Uses the X Portable Character Set.

**host-portable**

A synonym for **latin-portable**.

---

**false**

Function

Summary

Ignores its arguments and returns **nil**.

Package

lispworks

Signature

false &rest ignore => nil

Arguments

ignore

All arguments are ignored.

Value

nil

Description

The function **false** takes any number of arguments, which it ignores, and returns **nil**. It is useful as a functional argument.

See also

do-nothing
true

---

**file-directory-p**

Function

Summary

Tests for the presence of a directory.

Package

lispworks
**Signature**

file-directory-p pathname => bool

**Arguments**

pathname A pathname, string, or file-stream.

**Values**

bool If t, the pathname represented by pathname exists and is a directory. If nil, it either does not exist, or it is not a directory.

**Description**

file-directory-p tests whether the pathname represents a directory.

**Example**

CL-USER 70 > (file-directory-p "~")
T

CL-USER 71 > (file-directory-p ".login")
NIL

---

**find-regexp-in-string**

*Function*

**Summary**

Matches a regular expression.

**Package**

lispworks

**Signature**

find-regexp-in-string pattern string &key start end from-end case-sensitive brackets-limits => pos, len

find-regexp-in-string pattern string &key start end from-end case-sensitive brackets-limits => pos, len, brackets-limits-vector

**Arguments**

pattern A string or a precompiled regular expression object.

string A string.

start, end Bounding index designators of string.

from-end A generalized boolean.

case-sensitive A generalized boolean.

brackets-limits A generalized boolean.
Values

- \( pos \) A non-negative integer or nil.
- \( len \) A non-negative integer or nil.

brackets-limits-vector

A vector.

Description

The function \texttt{find-regexp-in-string} searches the string \texttt{string} for a match for the regular expression \texttt{pattern}. The index in \texttt{string} of the start of the first match is returned in \texttt{pos}, and the length of the match is \texttt{len}.

If \texttt{from-end} is nil (the default value) then the search starts at index \texttt{start} and ends at index \texttt{end}. \texttt{start} defaults to 0 and \texttt{end} defaults to nil. If \texttt{from-end} is true, then the search direction is reversed.

\texttt{pattern} should be a precompiled regular expression object or a string. If \texttt{pattern} is a string then \texttt{find-regexp-in-string} first makes a precompiled regular expression object. This operation allocates, therefore if you need to repeatedly call \texttt{find-regexp-in-string} with the same pattern, it is better to call \texttt{precompile-regexp} once and pass its result, a precompiled regular expression object, as \texttt{pattern}.

\texttt{case-sensitive} controls whether a string \texttt{pattern} is precompiled as a case sensitive or case insensitive search. A non-nil value means a case sensitive search. The value nil (the default) means a case insensitive search. \texttt{case-sensitive} is ignored if \texttt{pattern} is not a string.

When \texttt{brackets-limits} is non-nil, a successful call to \texttt{find-regexp-in-string} returns a third value which is a vector specifying the limits of matches of any pair of \texttt{\} and \texttt{\)} in the search pattern. The length of the vector is twice the number of pairs, and the elements are offsets from the beginning of the match of the whole pattern. Each pair of \texttt{\} (and \texttt{\)} is assigned a number in the order of the appearance of the \texttt{\)} in the pattern. This number multiplied by two is the index into the vector where the match for this pair starts, and the next
element specifies the end of the match. The default value of `brackets-limits` is `nil`.

The regular expression syntax used by `find-regexp-in-string` is similar to that used by Emacs, as described in the "Regular expression syntax" section of the *LispWorks Editor User Guide*. (If you use Help > Search to locate this section, select the Contents radio button.)

**Example**

This form allocates several regular expression objects:

```lisp
(loop with pos = 0
      with len = 0
      while pos
        do (multiple-value-setq (pos len)
            (find-regexp-in-string "[0,2,4,6,8]"
                         "0123456789"
                         :start (+ pos len)))
        when pos
          do (format t "Match at pos ~D len ~D~%" pos len))

This form does the same matching but allocates just one pre-compiled regular expression object:

```lisp
(loop with pattern = (precompile-regexp "[0,2,4,6,8]"")
      with pos = 0
      with len = 0
      while pos
        do (multiple-value-setq (pos len)
            (find-regexp-in-string pattern "0123456789" :start (+ pos len)))
        when pos do (format t "Match at pos ~D len ~D~%" pos len))
```

See also

- `precompile-regexp`
- `regexp-find-symbols`
- `count-regexp-occurrences`

**function-lambda-list**

**Summary**

Returns the argument list of the given function.
Package  lispworks

Signature  function-lambda-list  function &optional  error-p  =>  args

Arguments  function  A symbol or a function.
            error-p  A boolean.

Values  args  A list, or the symbol  :none

Description  function  is  the  function  whose  arguments  are  required
If  error-p  is  nil,  then  function-lambda-list  returns  :none
            if  function  is  not  defined,  and  does  not  start  the  debugger.
The  default  value  of  error-p  is  t,  meaning  that  an  error  is  sig-
naled  if  function  is  undefined.

Example  TEST 2 > (function-lambda-list 'editor:create-buffer-
            command)
            (EDITOR::P  &OPTIONAL  EDITOR:BUFFER-NAME)

get-inspector-values
sort-inspector-p

Generic Functions

Summary  Customizes  the  information  displayed  and  sort  order  of
attributes/values  in  the  LispWorks  IDE  Inspector  tool.

Package  lispworks

Signature  get-inspector-values  object  mode  =>  names,  values,  getter,
            setter,  type

Signature  sort-inspector-p  object  mode  =>  result

Arguments  object  The  object  to  be  inspected.
### Description

The generic functions `get-inspector-values` and `sort-inspector-p` allow you to customize the LispWorks IDE Inspector tool by adding new ways to display class instances and control sorting of the attributes and values.

`get-inspector-values` allows you to add new formats (corresponding to different values of mode) in which instances of a particular class can be inspected. Mode `nil` is the default mode, which is always present (it can be overwritten).

LispWorks includes methods for:

```
(get-inspector-values (object nil))
(get-inspector-values (standard-object nil))
(get-inspector-values (structured-object nil))
(get-inspector-values (sequence nil))
(get-inspector-values cons nil))
```

and so on.

`sort-inspector-p` determines whether to sort the list of displayed attributes/values.

The Inspector tool calls `sort-inspector-p` with the current object and mode the first time it displays this object in this mode to determine whether to sort the list of attributes/values. If it returns non-nil, it sorts by item, otherwise it does not sort.

### Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mode</code></td>
<td>Name of a mode, or <code>nil</code>. <code>nil</code> defines the default inspection format for <code>object</code>.</td>
</tr>
<tr>
<td><code>names, values</code></td>
<td>The two lists displayed in columns in the Inspector window.</td>
</tr>
<tr>
<td><code>getter</code></td>
<td>Ignored.</td>
</tr>
<tr>
<td><code>setter</code></td>
<td>A function used to update slot values.</td>
</tr>
<tr>
<td><code>type</code></td>
<td>Displayed in the Inspector window.</td>
</tr>
<tr>
<td><code>result</code></td>
<td>A boolean.</td>
</tr>
</tbody>
</table>
There are various methods on system-defined types to get the most useful behavior. You can add methods for your own types.

Notes
The sort type can be changed interactively in the Inspector tool by using the Preferences... dialog.

Example
This example allows inspection of a CLOS object, displaying only direct slots form a chosen class in its class precedence list. This can be useful when an object inherits many slots from superclasses, and the inherited slots are of no interest.
(defmethod lispworks:get-inspector-values
  ((object standard-object)
   (mode (eql 'direct-as)))
  (declare (ignore mode))
  (loop with object-class =
        (class-of object)
        with precedence-list =
        (class-precedence-list object-class)
        with items =
        (loop for super in precedence-list
              collecting (list*
                          (format nil "~-a"
                                  (class-name super))
                          super))
        with class =
        (or (capi:prompt-with-list items
                        "Direct slots as ...")
            object-class)
        ;; default if no selection
        with slots =
        (class-direct-slots class)
        for slot in slots
        for name =
        (clos::slot-definition-name slot)
        collect name into names
        collect (if (slot-boundp object name)
                  (slot-value object name)
                  :slot-unbound)
        into values
  finally
  (return
   (values
    names
    values
    nil
    #'(lambda
       (x slot-name index new-value)
       (declare (ignore index))
       (setf (slot-value x slot-name)
             new-value))
       (format nil "~-a - direct slots as -a"
               (class-name object-class)
               (class-name class))))))
get-unix-error

Function

Summary
Returns the text associated with a given error.

Package
lispworks

Signature
get-unix-error number => error

Arguments
number
The_errno value whose text is required.

Values
error
The text associated with the error.

Description
The function get-unix-error returns the text associated with the specified value of the UNIX variable errno.

Notes
get-unix-error is implemented only on non-Windows platforms.

See also
erno-value

*grep-command*

Variable

Package
lispworks

Summary
Determines the search utility used by Grep searches in the Search Files tool in the LispWorks IDE.

Initial value
"grep" on non-Windows platforms.
nil on Windows.

Description
If the value is a string, it is the search utility to run in the Search Files tool.
If the value is nil, then the value of (sys:lispworks-file "etc/grep")
is expected to be an executable, which is run. On Windows a suitable `grep.exe` is included with LispWorks in this location.

The search utility is passed arguments constructed using `*grep-command-format*` and `*grep-fixed-args*`.

See the LispWorks IDE User Guide for more information about the Search Files tool.

See also `*grep-command-format*`  
`*grep-fixed-args*`

### `*grep-command-format*`  

**Package**  
lispworks

**Summary**  
The format string used to construct the arguments passed to the Search Files tool to perform a Grep search.

**Initial value**  
"cd '-a'; -a -a -a /dev/null" on non-Windows platforms.  
"-a -a -a NUL" on Windows.

**Description**  
On non-Windows platforms, the first format argument is the current directory.

The remainder of the format arguments are:

- the value of `*grep-command*` or, if this is nil, the value of `(sys:lispworks-file "etc/grep")`.
- the value of `*grep-fixed-args*`.
- the arguments you specify.

See the LispWorks IDE User Guide for more information about the Search Files tool.
See also  
*grep-command*

*grep-fixed-args*

**Variable**

**grep-fixed-args**

**Package** lispworks

**Summary** Arguments added to the command string of a Grep search in the Search Files tool.

**Initial value** "-n"

**Description** The variable *grep-fixed-args* provides arguments added to a Grep command string in the Search Files tool. The value should ensure that the line number is output at the start of each match.

See the *LispWorks IDE User Guide* for more information about the Search Files tool.

See also  
*grep-command*

*grep-command-format*

**Variable**

**handle-existing-action-in-action-list**

**Summary** Contains keywords determining behavior on exceptions raised when an action definition already exists in a given action list.

**Package** lispworks

**Initial value** (:warn :redefine)

**Description** The variable *handle-existing-action-in-action-list* is a list containing one of :warn, or :silent, determining whether to notify the user, and one of :skip, or :redefine,
to determine what to do about an action definition when the action already exists in the given action list.

It is used by define-action.

See also define-action

*handle-existing-action-list*  
Variable

Summary Contains keywords determining what to do about a given action list operation when the action list already exists.

Package lispworks

Initial value (:warn :skip)

Description A list containing either :warn or :silent, determining whether to notify the user, and either :skip or :redefine to determine what to do about an action list operation when the action list already exists. The initial value is (:warn :skip).

It is used by the macro define-action-list.

See also define-action-list

*handle-missing-action-list*  
Variable

Summary Defines how to handle an operation on a missing action list.

Package lispworks

Signature *handle-missing-action-list*

Initial value :error
The variable `*handle-missing-action-list*` is a keyword; one of `:warn`, `:error`, or `:ignore`, denoting how to handle an operation on a missing action-list. The default value is `:error`.

`*handle-missing-action-list*` is used by `undefine-action-list`, `print-actions`, `execute-actions`, `define-action` and `undefine-action`.

See also `define-action`  
`execute-actions`  
`print-actions`  
`undefine-action`  
`undefine-action-list`

**Variable**

*handle-missing-action-in-action-list*  

Summary Denotes how to handle an operation on a missing action.

Package `lispworks`

Initial value `:warn`

Description The variable `*handle-missing-action-in-action-list*` is a keyword; one of `:warn`, `:error` or `:ignore`, denoting how to handle an operation on a missing action. Its initial value is `:warn`. It is used by `undefine-action`.

See also `undefine-action`

*handle-warn-on-redefinition*  

Summary Specifies the action on defining a symbol in certain packages.

Package `lispworks`
Initial value  :error

Description  The variable *handle-warn-on-redefinition* specifies what action should be taken on defining external symbols in certain packages. It is designed to protect against (re)definition of symbols in implementation packages.

The protected packages are those specified in the variable *packages-for-warn-on-redefinition*.

If *handle-warn-on-redefinition* is set to :warn then you are warned. If it is set to :quiet or nil, the definition is done quietly. If, however, it is set to :error, then LispWorks signals an error.

Notes  The checking is useful because it is relatively easy to redefine an external symbol by mistake, and it leads to undefined behavior which is difficult to debug. It is therefore a bad idea to change the value of *handle-warn-on-redefinition* to something else. If required, do this by rebinding *handle-warn-on-redefinition* rather than setting its global value.

See also  *packages-for-warn-on-redefinition*
*redefinition-action*
“Protecting packages” on page 88

**hardcopy-system**

*Function*

Summary  Print each file of a system to a printer.

Package  lispworks

Signature  hardcopy-system system-name &key command simulate => nil

Arguments  system-name  A symbol representing the name of the system. The system must have been defined using the defsystem macro.
simulate  If \texttt{nil} or not present then \texttt{hardcopy-system} works silently. Otherwise a plan of the actions which \texttt{hardcopy-system} intends to carry out is printed. What happens next depends on the value of \texttt{simulate}:

\begin{itemize}
  \item \texttt{t} — do nothing.
  \item \texttt{:ask} — you are asked, using \texttt{y-or-n-p}, if you want the plan to be carried out.
  \item \texttt{:each} — \texttt{hardcopy-system} displays each action in the plan one at a time, and asks you if you want to carry out this particular action. The answer executes the rest of the plan without further prompting, \texttt{e} returns from \texttt{hardcopy-system} without further processing, and \texttt{y} and \texttt{n} work as expected.
\end{itemize}

Values  \texttt{hardcopy-system} returns \texttt{nil}.

Examples  \begin{verbatim}
(hardcopy-system 'blackboard)
(hardcopy-system 'tms :simulate :ask :command "lpr")
\end{verbatim}

Notes  By default, \texttt{hardcopy-system} uses \texttt{*print-command*} as the command sent to the shell.

See also  \texttt{defsystem}, \texttt{*print-command*}

\texttt{*init-file-name*}  \textit{Variable}

Summary  The default user initialization file.

Package  \texttt{lispworks}

Initial value  \texttt{"~/.lispworks"}
The variable `*init-file-name*` is the name of the default user initialization file.

However, if the user initialization file is specified by either:

- the command line argument `-init`, or
- user preferences (as set via the Preferences dialog in the LispWorks IDE)

then the value of `*init-file-name*` is not used.

---

The variable `*inspect-through-gui*` controls what `inspect` does in the development environment.

When the value is `nil`, `inspect` uses a command line interface in the REPL.

When the value is `true`, `inspect` invokes an Inspector tool in the LispWorks IDE.

---

`lisp-image-name` returns the name of the running image.

**Signature**

`lisp-image-name => name`
The function lisp-image-name returns a string representing the full path to the running LispWorks image. The example below is in typical LispWorks for Windows and LispWorks for Linux installations. In resaved and delivered images (including dynamic libraries such as Windows DLLs), the appropriate path is returned.

Example

On Windows:

CL-USER 1 > (lisp-image-name)
"C:\Program Files\LispWorks\lispworks-7-1-0-x86-win32.exe"

On Linux:

CL-USER 1 > (lisp-image-name)
"/usr/bin/lispworks-7-1-0-x86-linux"

See also

*line-arguments-list*

*lispworks-directory*  
Variable

Summary

The main LispWorks installation directory.

Package

lispworks

Initial value

Some examples of the initial value are:

#P"/usr/lib/lispworks/" on SPARC Solaris.

#P"/usr/local/lib/LispWorks/" on Linux (for an installation from the tar archive) x86/x64 Solaris or FreeBSD.

#P"/usr/lib64/LispWorks/" on Linux (for an RPM installation)
The variable \texttt{*lispworks-directory*} holds the name of the directory where various files important for the running of LispWorks are located.

When LispWorks starts in a directory which contains an appropriate numbered subdirectory such as \texttt{lib/7-1-0-0/}, then it assumes this is the LispWorks installation directory and sets \texttt{*lispworks-directory*} accordingly. Additionally, LispWorks for Macintosh running on Cocoa looks for such a subdirectory in the \texttt{Library} folder alongside its application bundle, and if found it sets \texttt{*lispworks-directory*} accordingly.

On non-Windows platforms, LispWorks then consults the Unix environment variable \texttt{LISPWORKS_DIRECTORY}. If this is set, then \texttt{*lispworks-directory*} is set accordingly.

The \texttt{lib/7-1-0-0/} subdirectory of \texttt{*lispworks-directory*} should include these subdirectories:

- \texttt{config}, which contains the configuration files.
- \texttt{patches}, which contains any public (numbered) patches that are distributed by LispWorks Ltd.
- \texttt{private-patches}, which is the place to put private (named) patches that are sent to you by Lisp Support.
- \texttt{postscript}, which contains configuration files for printing using the CAPI printing library. See “Configuring the printer” on page 189 for more information on printer configuration.
examples, which contains various files of example code.

Other directories are etc, load-on-demand and manual. There is also app-defaults for platforms where Motif is supported.

load-all-patches

Function

Summary Loads all patch files into the image.

Package lispworks

Signature load-all-patches => nil

Arguments None.

Values Returns nil.

Description Loads into the image all appropriate files from the directory patches in the directory determined by *lispworks-directory*, and then loads the file private-patches/load.lisp where load forms for any private patches may be placed. When the appropriate patches have successfully been loaded, the updated version of the image can be saved using save-image.

You should call load-all-patches before starting the LispWorks IDE. Thus, you normally place the call to this function in your .lispworks file.

The system expects all patches to be loaded sequentially. If a patch is missing, there is a warning message. In this situation, it is advisable to contact Lisp Support to obtain a copy of the missing patch.
**load-system**

**Function**

**Summary**
Load each file of a system into the Lisp image if either the file has not been loaded, or the file has been written since it was last loaded.

**Package**
lispworks

**Signature**
load-system system-name &key force simulate source-only target-directory => nil

**Arguments**

- **system-name**
  A symbol representing the name of the system. The system must have been defined using the defsystem macro.

- **force**
  If t then all the files in the system are loaded regardless. (This argument was formerly called force-p. The old name is currently still accepted for compatibility.)

- **simulate**
  If nil or not present then load-system works silently. Otherwise a plan of the actions which load-system intends to carry out is printed. What happens next depends on the value of simulate:

  - t — do nothing.
  - :ask — you are asked, using y-or-n-p, if you want to carry out the plan.
  - :each — load-system displays each action in the plan one at a time, and asks you if you want to carry out this particular action. The answer executes the rest of the plan without further prompting, e returns from load-system without further processing, and y and n work as expected.
source-only  If t the source files of the system are loaded. This only applies to file types where it makes sense to load a source file.

target-directory  This is the directory to search for the object files. If the object file cannot be found here then the source file from the system’s default directory are loaded.

Examples  

(load-system 'blackboard)

(load-system 'tms :simulate :ask :source-only t)

Notes  For Lisp files load-system loads the object file (if it exists) into the image, unless over-ridden by the :source-only keyword argument. This behavior can be changed so that the newest file (whether source or object) is loaded by setting the variable *load-source-if-newer* to t.

C source files, for example foo.c, can be included in a system (see the use of :default-type and :type in defsystem). The corresponding object file name is foo.so on Linux, FreeBSD, AIX and x86/x64 Solaris, foo.dylib on Mac OS X and foo.o on SPARC Solaris, where n is a platform-specific integer. On Windows the object file name is foo.dll.

See also  defsystem  compile-system  concatenate-system

make-unregistered-action-list  Function

Summary  Makes an unregistered action list.

Package  lispworks
Signature  
\texttt{make-unregistered-action-list \&key documentation sort-time dummy-actions default-order execution-function =>}

Arguments  
- \textit{documentation}  A string.
- \textit{sort-time}  One of \texttt{:execute} or \texttt{:define-action}.
- \textit{dummy-actions}  A list.
- \textit{default-order}  A list.
- \textit{execution-function}  A function.

Description  
Return an action-list not registered in the global registry of lists. The keyword arguments are as for \texttt{define-action-list}.

The \textit{documentation} string allows you to provide documentation for the action list.

\textit{sort-time} is a keyword specifying when added actions are sorted for the given list — either \texttt{:execute} or \texttt{:define-action} (see \texttt{*default-action-list-sort-time*}).

\textit{dummy-actions} is a list of action-names that specify placeholder actions; they cannot be executed and are constrained to the order specified in this list, for example

'(beginning :middle :end)

default-order specifies default ordering constraints for subsequently defined action-items where no explicit ordering constraints are specified. An example is

'(after :beginning :before :end)

\textit{execution-function} specifies a user-defined function accepting arguments of the form:

\texttt{(the-action-list other-args-list \&rest keyword-value-pairs)}

where the two required arguments are the action-list and a list of additional arguments passed to \texttt{execute-actions}, respectively. The remaining arguments are any number of keyword-value pairs that may be specified in the call to
execute-actions. If no execution function is specified, then the default execution function will be used to execute the action-list.

See also

define-action-list
*handle-warn-on-redefinition*

make-mt-random-state

Summary

Creates an object of type mt-random-state.

Package

lispworks

Signature

make-mt-random-state &optional state => new-state

Arguments

state nil, t or an object of type mt-random-state. The default is nil.

Values

new-state A new object of type mt-random-state.

Description

The function make-mt-random-state creates a new object of type mt-random-state which is suitable for use as the value of *mt-random-state*.

If state is an object of type mt-random-state, then new-state is a copy of state. If state is nil, then new-state is a copy of the value of *mt-random-state*. If state is t then new-state is an object of type mt-random-state initialized using a call to get-universal-time.

make-mt-random-state is analogous to cl:make-random-state.

See also

mt-random
*mt-random-state* mt-random-state
**mt-random**

*Function*

<table>
<thead>
<tr>
<th>Summary</th>
<th>Returns a pseudo-random number using the Mersenne Twister algorithm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>lispworks</td>
</tr>
<tr>
<td>Signature</td>
<td>mt-random arg &amp;optional state =&gt; random-number</td>
</tr>
<tr>
<td>Arguments</td>
<td>arg</td>
</tr>
<tr>
<td></td>
<td>state</td>
</tr>
<tr>
<td>Values</td>
<td>random-number</td>
</tr>
<tr>
<td>Description</td>
<td>The function mt-random returns a pseudo-random number which is non-negative, less than arg and is of the same type as arg. random-number is generated using the Mersenne Twister algorithm published by Makoto Matsumoto and Takuji Nishimura at <a href="http://www.math.keio.ac.jp/~matumoto/emt.html">http://www.math.keio.ac.jp/~matumoto/emt.html</a>. We thank the authors for making the algorithm freely available. mt-random is analogous to cl:random.</td>
</tr>
<tr>
<td>See also</td>
<td>make-mt-random-state</td>
</tr>
<tr>
<td></td>
<td><em>mt-random-state</em></td>
</tr>
</tbody>
</table>

**Variable**

<table>
<thead>
<tr>
<th>Summary</th>
<th>The default random state used by mt-random.</th>
</tr>
</thead>
</table>

*mt-random-state*
The variable *mt-random-state* contains an object of type mt-random-state which is the default state used by mt-random if a state is not supplied. *mt-random-state* is analogous to cl:*random-state*.

See also make-mt-random-state

mt-random

mt-random-state

The type of objects containing state information used by mt-random.

See also *mt-random-state*

mt-random

mt-random-state-p

The predicate for objects of type mt-random-state.

Signature mt-random-state-p arg => result
Arguments  

arg  

An object.

Values  

result  

A boolean.

Description  

The function `mt-random-state-p` returns `t` if `arg` is an object of type `mt-random-state`, and `nil` otherwise.

`mt-random-state-p` is analogous to `cl:random-state-p`.

See also  

`mt-random-state`

### pathname-location

**Function**

Summary  

Returns the location of a file.

Signature  

`pathname-location pathname => location`

Arguments  

pathname  

A pathname designator.

Values  

location  

A pathname.

Description  

The function `pathname-location` returns a pathname `location` that represents the directory where the file `pathname` resides. Each of the name, type and version components of `location` are `nil`.

Example  

Due to the ANSI Common Lisp definition of the `directory` function and the fact that LispWorks returns fully specified truenames, the form

```
(directory (truename "/tmp/"))
```

will always signal an error or return the list `(#P"/tmp")`. To obtain the contents of the `/tmp` directory, use the form

```
(directory (pathname-location (truename "/tmp/")))
```
See also  
  current-pathname
  directory

**precompile-regexp**

**Function**

**Summary**
Precompiles a regular expression object.

**Package**
lispworks

**Signature**
`precompile-regexp string &key case-sensitive error-function => pattern, condition-designators`

**Arguments**
- `string` A string.
- `case-sensitive` A generalized boolean.
- `error-function` nil or a function that takes arguments like `error`.

**Values**
- `pattern` A precompiled regular expression object.
- `condition-designators` - A list.

**Description**
The function `precompile-regexp` returns a precompiled regular expression object suitable for passing as `pattern` to functions like `find-regexp-in-string`.

`case-sensitive` controls whether `string` is precompiled as a case sensitive or case insensitive search. A non-nil value means a case sensitive pattern. The value nil (the default) means a case insensitive pattern.

`error-function` is used when the string is not a legal regular expression. In this case, if `error-function` is not nil, it is applied to a list of arguments which are designators for a condition like the arguments that `error` takes. If `error-function` is nil, `precompile-regexp` returns nil as the first argument and the list of arguments as a second return value, `condition-designators`. `error-function` defaults to `error`. 
For the regular expression syntax, see `find-regexp-in-string`.

See also `find-regexp-in-string`
`regexp-find-symbols`
`count-regexp-occurrences`
`editor:regular-expression-search`

---

### `print-actions` Function

**Summary**
Prints a listing of the action items on a given action list in order.

**Package**
lispworks

**Signature**
`print-actions name-or-list &optional stream`

**Arguments**

- `name-or-list` An action list.
- `stream` An output stream.

**Description**
The function `print-actions` prints a listing of the action items on the action-list denoted by `name-or-list`, in order.

If the action-list specified by `name-or-list` does not exist, then this is handled according to the value of `*handle-missing-action-list*`.

The output is written to the stream `stream`. The default value of `stream` is the value of `*standard-output*`.

**See also**
`print-action-lists`

### `print-action-lists` Function

**Summary**
Prints a list of all the action lists in the global registry.
Package lispworks

Signature print-action-lists &optional stream

Arguments stream An output stream.

Description The function print-action-lists prints a listing of all the action lists in the global registry. The ordering of the action lists is random.

The output is written to the stream stream. The default value of stream is the value of *standard-output*.

See also print-actions

*print-command* Variable

Summary A command used for some printing operations.

Package lispworks

Initial value "print" on Windows.

"lpr" on Mac OS X and all Unix-like systems.

Description This variable is used as the command sent by LispWorks to the shell in hardcopy-system.

See also hardcopy-system

*print-nickname* Variable

Summary Controls the package prefix used when a symbol is printed.

Package lispworks
Initial value  

nil

Description  
The variable *print-nickname* controls which package prefix is used when a symbol is printed and the symbol’s package needs to be output.

If *print-nickname* is true and the package has at least one nickname, then the first of the nicknames (that is, the first nickname in the list returned by package-nicknames) is output. Otherwise, the package name is output.

*prompt*  

Variable

Summary  
Defines the LispWorks listener prompt.

Package  
lispworks

Initial value  
"~%~A ~D~[~:;~:* : ~D~] > "

Description  
The variable *prompt* defines the LispWorks listener prompt. Its value can be a:

Function designator  
A function of zero arguments which should return the prompt as a string.

String  
A format string with processing three arguments: the current package name, the next history number, and the debug level.

A form  
The form is passed to eval and should return a format string, which is used as for the string case above.
Example

CL-USER 1 > (defvar *default-prompt* *prompt*)
*DEFAULT-PROMPT*

CL-USER 2 > (progn
(setf *prompt*
  '(string-append "-&"
    (sys:get-user-name)
    #\Space
    (subseq *default-prompt* 2)))
  nil)
NIL

Example

push-end
push-end-new

Macros

Summary
Append an item to a list stored in a place.

Package
lispworks

Signature
push-end item place => new-place-value
push-end-new item place &key key test test-not => new-place-value

Arguments
item Anything.
place A generalized reference form as described in section 5.1.1 Overview of Places and Generalized Reference of the Common Lisp Hyperspec.
key, test, test-not Function designators.

Values
new-place-value A list which is the new value of place.
The macros **push-end** and **push-end-new** are analogs to **push** and **pushnew**, except that they append **item** to the end of the list rather than prepend it.

**place** must contain a proper list.

**push-end** sets **place** to a copy of this list with **item** appended in the end.

**push-end-new** does the same as **push-end**, except when **item** is already on the list, in which case **push-end-new** does nothing. The check is done using the values of **key**, **test** and **test-not** in the same way that **pushnew** does.

The return value **new-place-value** is the value of **place** after the operation. Except when **item** is already in the list, it is always a new list.

**Notes: Multi-threading**

**push-end** and **push-end-new** are not atomic.

If **place** is globally accessible and may be read by another thread without synchronization (by a **lock** or other synchronization mechanism), then you need to wrap **place** by **globally-accessible**, for example:

```lisp
(push-end my-item
  (sys:globally-accessible
    *a-global-symbol*))
```

See “Making an object’s contents accessible to other threads” on page 268 for a discussion.

**push** and **pushnew** also have the same issues with Multi-threading.

---

**quit**

**Function**

**Summary** Quits LispWorks.

**Package** lispworks
The function `quit` exits LispWorks unless the user cancels the operation.

There are two stages which may allow the user the chance to cancel.

1. First the action items of the action list "Confirm when quitting image" are run. If any action item returns `nil`, then LispWorks does not exit.

2. Otherwise, if `confirm` is true (the default value is `nil`) then a question like "Do you really want to exit LispWorks?" is presented to the user. If the answer No is supplied, then LispWorks does not exit. Otherwise, the action items of the action list "When quitting image" are run, and then LispWorks exits, and the value `status` is returned to the Operating System as the exit value of the LispWorks process. The default value of `status` is 0.

If `ignore-errors-p` is true, then any error signaled during the running of the action list items or the confirm prompt is ignored and `quit` proceeds to exit the image. If `ignore-errors-p` is `nil` and an error is signaled during the running of the action list items, then a restart is available allowing the user to choose to continue to exit the image. The default values of `ignore-errors-p` is `nil`.

If `return` is true and LispWorks is going to exit, then `quit` returns `t`. This can be used if you want some other Lisp pro-
cess to kill the current one later, rather than it self-destructing immediately. This can be useful to allow more precise control over process termination. If return is nil then quit does not return. The default value of return is nil.

Notes
On Cocoa, when you define your own application menu (by passing :application-menu when making the application interface), the Quit menu item needs to call capi:destroy on the application interface, rather than quit. See capi:cocoa-default-application-interface in the CAPI User Guide and Reference Manual for more information.

See also save-image

rebinding

Macro

Summary Ensures unique names for all the variables in a group of forms.

Package lispworks

Signature re-binding (&rest vars) &body body => form

Arguments vars The variables to be rebound.
body A body of forms, the variables in which should be unique.

Values Returns the body wrapped in a form that creates unique names for each variable.

Description Returns the body wrapped in a form which creates a unique name for each of the variables (compare with gensym) and binds these names to the values of the variables. This ensures that the body can refer to the variables without name clashes with other variables elsewhere.
Example

After defining

```
(defun lister (x y)
  (rebinding (x y)
    '(list ,x ,y)))
```

the form `(lister i j)` macroexpands to

```
(LET* ((#:X-77 I)
       (#:Y-78 J))
  (LIST #:X-77 #:Y-78))
```

See also `with-unique-names`

**regexp-find-symbols**

*Function*

**Summary**

Returns a list of symbols that match a supplied regular expression.

**Package**

lispworks

**Signature**

`regexp-find-symbols regexp-string &key case-sensitive packages test external-only => symbols`

**Arguments**

- `regexp-string`: A string.
- `case-sensitive`: A boolean.
- `packages`: A list of package designators, a single package designator, or the keyword `:all`.
- `test`: A function of one argument returning a boolean result.
- `external-only`: A generalized boolean.

**Values**

- `symbols`: A list of symbols.

**Description**

The function `regexp-find-symbols` returns a list of symbols that match the regular expression in `regexp-string`. 
case-sensitive determines whether the match is case sensitive. The default value of case-sensitive is nil.

packages specifies in which packages to search. The default value of packages is :all, meaning search in all packages.

test, if supplied, must be a function of one argument, which returns t if the argument should be returned, and nil otherwise. The function test is applied to each symbol that matches regexp-string, and if it returns nil the symbol is not included in the returned value symbols. If test is nil all matches are returned. The default value of test is nil.

external-only, if true, specifies that only external symbols should be checked, which makes the search much faster. The default value of external-only is nil.

The regular expression syntax used by regexp-find-symbols is similar to that used by Emacs, as described in the "Regular expression syntax" section of the LispWorks Editor User Guide.

Examples

To find all exported symbols that start with DEF:

(lw:regexp-find-symbols "^def" :external-only t)

To find all symbols that contain lower case "slider":

(regexp-find-symbols "slider" :case-sensitive t)

See also

apropos
find-regexp-in-string

remove-advice

Function

Summary
Remove a piece of advice.

Package
lispworks
Signature  \texttt{remove-advice function-dspec name \Rightarrow nil}

Arguments  
\begin{itemize}
  \item \textit{function-dspec} \hspace{1cm} A function-dspec Specifies the function definition to which the piece of advice belongs. See “Function dspecs” on page 84 for description of function-dspec.
  \item \textit{name} \hspace{1cm} A symbol naming the piece of advice to be removed. Since several pieces of advice may be attached to a single functional definition, the name is necessary to indicate which one is to be removed.
\end{itemize}

Values  \texttt{remove-advice} returns \texttt{nil}.

Description  \texttt{remove-advice} is the function used to remove a piece of advice. Advice is a way of altering the behavior of functions. Pieces of advice are associated with a function using \texttt{defadvice}. They define additional actions to be performed when the function is invoked, or alternative code to be performed instead of the function, which may or may not access the original definition. As well as being attached to ordinary functions, advice may be attached to methods and to macros (in this case it is in fact associated with the macro’s expansion function).

\texttt{hcl:delete-advice} is a macro, identical in effect to \texttt{remove-advice}, except that you do not need to quote the arguments.

Notes  \texttt{remove-advice} is an extension to Common Lisp.

See also  \texttt{defadvice} \newline \texttt{delete-advice} \newline Chapter 6, “The Advice Facility”
**removef**

*Macro*

**Summary**
Removes an item from a sequence.

**Package**
lispworks

**Signature**

```
removef place item &key test test-not start end key => result
```

**Arguments**

- `place` A place.
- `item` An object.
- `test` A test function.
- `test-not` A test function.
- `start` An integer.
- `end` An integer or `nil`.
- `key` A key function.

**Values**

- `result` A sequence.

**Description**

The modifying macro `removef` removes an item from a sequence using `remove`. See `remove` for more details.

**See also**
appendf

**require-verbose**

*Variable*

**Summary**
Controls the output of `require`.

**Package**
lispworks

**Initial value**
t

**Description**

The variable `require-verbose` is a generalized boolean controlling whether `require` prints the names of the files which are being loaded.
**rotate-byte**  
*Summary* Rotates specified bits within an integer.  
*Package* lispworks  
*Signature*  
\[ \text{rotate-byte} \quad \text{count} \quad \text{byte-spec} \quad \text{integer} \quad \Rightarrow \quad \text{result-integer} \]  
*Arguments*  
- \text{count} An integer.  
- \text{byte-spec} A byte specifier.  
- \text{integer} An integer.  
*Values* \text{result-integer} An integer.  
*Description* The function \text{rotate-byte} returns \text{integer} with the bits specified by \text{byte-spec} rotated left by \text{count} bits. Other bits remain the same as in \text{integer}. If \text{count} is negative, then the effect is to rotate right.  
*See also* [http://www.cliki.net/rotate-byte](http://www.cliki.net/rotate-byte)  
*Examples*  
\[
(\text{rotate-byte} \ 2 \ (\text{byte} \ 3 \ 1) \ 99) \Rightarrow 105
\]
\[
(\text{rotate-byte} \ -2 \ (\text{byte} \ 3 \ 1) \ 99) \Rightarrow 101
\]

**round-to-single-precision**  
*Summary* Rounds the given float to single-precision format (32 bits) and returns it as a \text{double-float} (64 bits).  
*Package* lispworks  
*Signature*  
\[ \text{round-to-single-precision} \quad \text{float} \quad \Rightarrow \quad \text{double-float} \]  
*Arguments* \text{float} A float
Values

| double-float | A double-float with single-float precision. |

Description

The argument is rounded to single-precision format (32 bits) and returned as a **double-float** (64 bits). This function allows you to model the rounding behavior of a machine or implementation that performs 32-bit floating point arithmetic.

LispWorks supports multiple floating point formats: short-float (only on 32-bit LispWorks), single-float and double-float. If this function is called with a single-float or a short-float, it returns the equivalent double-float, that is, it is the same as evaluating

(coerce float 'double-float)

Compatibility notes

LispWorks 4.4 and previous on Windows and Linux platforms supports just one floating point format. In LispWorks 5.0 and later, at least two floating point formats are supported on all platforms.

Example

```
CL-USER 197 > pi
3.141592653589793D0

CL-USER 198 > round-to-single-precision pi
3.1415927410125732D0
```

**sbchar**

Function

Summary

The accessor for simple base strings.

Package

lispworks

Signature

`sbchar string index => value`

Arguments

| string     | A simple-base-string. |
| index      | An index. |
### Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string value</td>
<td>The character in string at index.</td>
</tr>
</tbody>
</table>

### Description

This is the accessor for simple base strings. `setf` is allowed.

### See also

`simple-base-string`

---

### sequencep

**Function**

**Summary** A predicate to check for sequences.

**Package** lispworks

**Signature**

sequencep object => result

**Arguments**

- object A Lisp object.

**Values**

- result A generalized boolean.

**Description**

The function `sequencep` returns true if `object` is of type `sequence` and false otherwise.

**Examples**

- `(sequencep '(1 2 3)) => t`
- `(sequencep #(1 2 3)) => t`
- `(sequencep 123) => nil`

---

### set-default-character-element-type

**Function**

**Summary** Configures the value of `lw:*default-character-element-type*`.

**Package** lispworks

**Signature**

set-default-character-element-type type => type-defaults
Arguments  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A character type. This can take any of the values <code>cl:base-char</code>, <code>bmp-char</code> and <code>cl:character</code>. For backwards compatibility, <code>simple-char</code> is also allowed, and is treated as if <code>cl:character</code> was passed.</td>
</tr>
</tbody>
</table>

Values  

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type-defaults</td>
<td>The new value of <code>lw:*default-character-element-type*</code>.</td>
</tr>
</tbody>
</table>

Description  

The function `set-default-character-element-type` sets the value of `lw:*default-character-element-type*`, ensuring that the system’s internal state is also updated accordingly.

If you are running an existing 8-bit application you will only need to have this in your site or user configuration file:

```
(lw:set-default-character-element-type 'base-char)
```

It would be a mistake to call this function in a loadable package and it is not intended to be called while running code. In particular, it is global, not thread-specific.

Hence we consider `lw:*default-character-element-type*` a parameter.

Compatibility note:  

`simple-char` is deprecated. Its meaning has changed between LispWorks 6 and 7.

See also  

- `string`
- `simple-string`
- `make-string`
- `open`
- `*default-character-element-type*`  
- `with-output-to-string`
- “Unicode support” on page 435
- “Controlling string construction” on page 442
set-quit-when-no-windows

Function

Summary
Overrides the :quit-when-no-windows keyword argument to deliver.

Signature
set-quit-when-no-windows on

Arguments
on nil, t or the keyword :check

Description
The function set-quit-when-no-windows can be used at run time in a delivered application to override the value of the :quit-when-no-windows keyword to deliver. This can be useful if the application runs in various modes, some with windows and some without. It has no effect in a non-delivered application.

If on is nil, then the application will not quit merely because there are no remaining open windows.

If on is t, then the application will quit when there are no remaining open windows after the application has opened at least one CAPI window.

If on is :check, then the application will quit immediately if there are no open windows at the current time. Unlike with :quit-when-no-windows t, this occurs even if the application has not opened any CAPI windows so far. If there are open windows currently, then it turns on quitting like when on is t.

See also :quit-when-no-windows in the LispWorks Delivery User Guide

simple-char

Type

Summary
The simple character type. simple-char is deprecated.

Package
lispworks
The type of simple characters (standard term for chars with null implementation-defined attributes, that is, no bits).

`simple-char` is a synonym for `cl:character`, and is deprecated.

Notes

16-bit characters and 16-bit strings are implemented by the types `bmp-char` and `bmp-string` and `simple-bmp-string`.

### `simple-char-p` Function

**Summary**
The predicate for simple characters. `simple-char-p` is deprecated.

**Package**
`lispworks`

**Signature**
`simple-char-p object => result`

**Arguments**
`object` The object to be tested.

**Values**
`result` A boolean.

**Description**
The function `simple-char-p` is the predicate for simple characters.

`result` is `t` if `object` is a simple character, and `nil` otherwise.

**See also**
`simple-char`

### `split-sequence` Function

**Summary**
Returns a list of subsequences of a sequence, split at specified separator elements.
Package  lispworks

Signature  split-sequence  separator-bag  sequence  &key  start  end  test  key  
            coalesce-separators  =>  sequences

Arguments  separator-bag  A sequence.
           sequence  A sequence.
           start, end  Bounding index designators for sequence.
           test  A function designator.
           key  A function designator or nil.
           coalesce-separators  A generalized boolean.

Values  sequences  A list of sequences.

Description  The function split-sequence returns a list of subsequences of sequence (between start and end), split when an element in the sequence separator-bag is found. The structure of sequence is not changed and the elements matching separator-bag are not included in the resulting sequences.

The function test, which defaults to eql, is used to compare the elements of sequence and the elements of separator-bag.

If true, the function key, is applied to the elements of sequence before test is called.

If coalesce-separators is true, then empty sequences are removed.

See also  split-sequence-if

split-sequence-if  Function

Summary  Returns a list of subsequences of a sequence, split at elements for which a predicate returns true.
Package lispworks

Signature

\texttt{split-sequence-if \textit{predicate} \textit{sequence} \&\textit{key} start end key coalesce-separators => result}

Arguments

\begin{itemize}
  \item \textit{predicate} \hspace{1cm} A function designator.
  \item \textit{sequence} \hspace{1cm} A sequence.
  \item \textit{start, end} \hspace{1cm} Bounding index designators for \textit{sequence}.
  \item \textit{key} \hspace{1cm} A function designator or \texttt{nil}.
  \item \textit{coalesce-separators} \hspace{1cm} A generalized boolean.
\end{itemize}

Values \textit{result} \hspace{1cm} A list of sequences.

Description

The function \texttt{split-sequence-if} returns a list of sub-sequences of \textit{sequence} (between start and end), split by where the function \textit{predicate} returns true for an element. The structure of \textit{sequence} is not changed and the elements identified by the predicate are not included in the resulting sequences. If non-nil, the function \textit{key} is applied to the elements of \textit{sequence} before \textit{predicate} is called.

If \textit{coalesce-separators} is true, then empty sequences are omitted from \textit{result}.

See also \texttt{split-sequence} \texttt{split-sequence-if-not}

\textbf{split-sequence-if-not} \hspace{1cm} \textit{Function}

Summary

Returns a list of subsequences of a sequence, split at elements for which a predicate returns false.

Package lispworks
Signature: `split-sequence-if-not predicate sequence &key start end key coalesce-separators => sequences`

Arguments:
- `predicate`: A function designator.
- `sequence`: A sequence.
- `start, end`: Bounding index designators for `sequence`.
- `key`: A function designator or `nil`.
- `coalesce-separators`: A generalized boolean.

Values: `result`: A list of sequences.

Description: The function `split-sequence-if-not` returns a list of subsequences of `sequence` (between `start` and `end`), split by where the function `predicate` returns false for an element. The structure of `sequence` is not changed and the elements identified by the predicate are not included in the resulting sequences.

If non-nil, the function `key` is applied to the elements of `sequence` before `predicate` is called.

If `coalesce-separators` is true, then empty sequences are omitted from `result`.

See also: `split-sequence` `split-sequence-if`

**start-tty-listener**

Function

Summary: Starts a listener in the startup shell.

Package: `lispworks`

Signature: `start-tty-listener force => process`
Arguments

force

A generalized boolean.

Values

process

A listener process, or nil.

Description

The function start-tty-listener returns a process that runs a listener read-eval-print loop connected to *terminal-io*.

If force is nil, then start-tty-listener checks whether the default listener process is alive or if there is a live process with name "TTY Listener". If such a process exists, start-tty-listener simply returns nil and does not start a new process. If no such process exists, or if force was t, then start-tty-listener starts a new listener process named "TTY Listener", and returns it.

If a REPL with I/O through *terminal-io* (such as a REPL started by start-tty-listener) is in the debugger, then by default it blocks multiprocessing. This behavior is controlled by the value of *terminal-debugger-block-multiprocessing*.

See also

*terminal-debugger-block-multiprocessing*  

stchar

Function

Summary

The accessor for simple text strings.

Package

lispworks

Signature

stchar string index => value

Arguments

string A simple-text-string.

index An index.

Values

value The character in string at index.
Description  
This is the accessor for simple text strings. setf is allowed.

See also  
simple-text-string

string-append  

Function

Summary  
Constructs a single string from a number of strings.

Package  
lispworks

Signature  
string-append &rest strings => string

Arguments  
strings  
Any number of string designators.

Values  
string  
A string.

Description  
The function string-append takes any number of string designators and constructs a single string from them.

Each of the elements of the strings argument is first coerced into a string using the string function if it is not already a string.

string is a string of the "widest" type amongst strings. That is, the constructed string is of the same type as the argument with the largest element type.
Example

(readtable-case *readtable*)

=>

:UPCASE

(string-append "foo" 'bar)

=>

"fooBAR"

(type-of

(string-append

 (coerce "A" 'simple-base-string)

 (coerce "A" 'simple-text-string)

))

=>

SIMPLE-TEXT-STRING

See also

string-append*

---

**string-append***

*Function*

**Summary**

Constructs a single string from a list of strings.

**Package**

lispworks

**Signature**

string-append* strings => string

**Arguments**

strings

A list of string designators.

**Values**

string

A string.

**Description**

The function **string-append*** takes a list of string designators and constructs a single string from them.

Each of the elements of the strings argument is first coerced into a string using the function **string** if it is not already a string.

**string** is a string of the "widest" type amongst strings. That is, the constructed string is of the same type as the string with
the largest element type amongst those supplied in the argument.

Example

```lisp
(readtable-case *readtable*)
=> :UPCASE

(string-append* '("foo" bar))
=> "fooBAR"

(type-of (string-append* (list (coerce "A" 'simple-base-string) (coerce "A" 'simple-text-string))))
=> SIMPLE-TEXT-STRING
```

See also

string-append

---

### structurep

**Function**

**Summary** A predicate to check for structure objects.

**Package** lispworks

**Signature** `structurep object => result`

**Arguments**

- `object` A Lisp object.

**Values**

- `result` A generalized boolean.

**Description** The function `structurep` returns true if `object` is of type `structure-object` and false otherwise.

**Examples**

```lisp
(structurep #(1 2 3)) => nil
```

Given the definition:
(defstruct my-struct a)

then

(structurep (make-my-struct)) => t

but metaclasses are not structures so:

(structurep (find-class 'my-struct)) => nil

text-string

simple-text-string

Types

Summary
The text string types.

Package
lispworks

Signature
text-string length

simple-text-string length

Arguments
length

The length of the string (or *, meaning any).

Description
text-string is the type of strings that can hold any character, that is, (vector cl:character length). This is the string type that is guaranteed to always hold any character used in writing text (program text or natural language).

simple-text-string is the simple version of text-string, that is, the string itself is simple. Equivalent to:

(simple-vector cl:character length)

Notes
text-string uses 32 bits per character. Applications that use many strings and are very large, when they know they do not use the full Unicode range, can consider using base-string (up to 8 bits) or bmp-string (up to 16 bits) to reduce memory usage.
Compatibility note

In LispWorks 6.1 and earlier versions, text-string uses 16 bits per character.

See also

bmp-string
base-string
text-string-p

“Character and String types” on page 436

text-string-p
simple-text-string-p

Functions

Summary
The predicates for text strings.

Package
lispworks

Signature
text-string-p object => result
simple-text-string-p object => result

Arguments
object A Lisp object.

Values
result A boolean.

Description
The functions text-string-p and simple-text-string-p are the predicates for text strings and simple text strings respectively.

result is t if object is a text-string (or simple-text-string), and nil otherwise.

See also
text-string
simple-text-string

true

Function

Summary
Ignores its arguments and returns t.
true

## Arguments
- **ignore**
  - All arguments are ignored.

## Values
- **t**

## Description
The function `true` ignores all its arguments and returns `t`. It is useful as a functional argument.

## See also
- `do-nothing`
- `false`

### undefine-action

**Macro**

## Summary
Removes an action from a specified list.

## Package
- lispworks

## Signature
```
undefine-action name-or-list action-name =>
```

## Arguments
- **name-or-list**
  - A list or action list object.
- **action-name**
  - A general lisp object.

## Description
The macro `undefine-action` removes the action specified by `action-name` from the action list specified by `name-or-list`. If the action specified by `action-name` does not exist, then this is handled according to the value of `*handle-missing-action-in-action-list*`.

`name-or-list` is evaluated to give either a list UID (to be looked up in the global registry of lists) or an action list object. `action-name` is a UID (general lisp object, to be compared by `equalp`). It uniquely identifies this action within its list (as opposed to among all lists).
See also define-action

**undefine-action-list**  
*Macro*

Summary       Removes a given defined action list.

Package  lispworks

Signature    undefine-action-list uid =>

Arguments   uid  A lisp object.

Values      None.

Description  The macro `undefine-action-list` flushes the specified list (and all its action-items). If the action-list specified by `uid` does not exist, then handling is controlled by the value of the variable `*handle-missing-action-list*`.

See also define-action-list

**unicode-alpha-char-p**  
*Function*

Summary       Returns a value like `cl:alpha-char-p`, but using specified Unicode rules.

Package  lispworks

Signature    unicode-alpha-char-p char &key style => flag

Arguments   char  A character

style  A keyword

Values      flag  A generalized boolean
Description
The function `unicode-alpha-char-p` returns flag as true if char is an alphabetic character according to the Unicode rules specified by style.

The current implementation only supports one style:

`:general-category`

Use the "general category" for char in Unicode 6.3.0.

See also
`unicode-alphanumeric-p`
`unicode-both-case-p`

**unicode-alphanumeric-p**

*Function*

Summary
Returns a value like `cl:alphanumericp`, but using specified Unicode rules.

Package
lispworks

Signature
`unicode-alphanumericp char &key style => flag`

Arguments
`char`  A character
`style` A keyword

Values
`flag` A generalized boolean

Description
The function `unicode-alphanumericp` returns flag as true if char is alphanumeric according to the Unicode rules specified by style.

The current implementation only supports one style:

`:general-category`

Use the "general category" for char in Unicode 6.3.0.
See also  
*unicode-alpha-char-p*
*unicode-both-case-p*

**unicode-both-case-p**

*Function*

**Summary** Returns a value like `cl:both-case-p`, but using specified Unicode rules.

**Package** lispworks

**Signature** `unicode-both-case-p char &key style => flag`

**Arguments**
- `char` A character
- `style` A keyword

**Values** `flag` A generalized boolean

**Description** The function `unicode-both-case-p` returns `flag` as true if `char` has case according to the Unicode rules specified by `style`.

The current implementation only supports one style:

`:general-category`

Use the "general category" for `char` in Unicode 6.3.0.

**Notes** The name of `unicode-both-case-p` is slightly confusing, because it matches the ANSI Common Lisp definition "a character with case" whereas there is no guarantee that both cases actually exist. Note also that there are some "alpha" chars which are not lower or upper case.

See also  
*unicode-alpha-char-p*
*unicode-lower-case-p*
*unicode-upper-case-p*
**unicode-char-equal**

**unicode-char-not-equal**

*Functions*

**Summary**

Compares two characters, ignoring case using specified Unicode rules.

**Package**

lispworks

**Signatures**

`unicode-char-equal char1 char2 &key style => flag`

`unicode-char-not-equal char1 char2 &key style => flag`

**Arguments**

`char1` A character

`char2` A character

`style` A keyword

**Values**

`flag` A generalized boolean

**Description**

The function `unicode-char-equal` returns true if the characters `char1` and `char2` are equal, and the function `unicode-char-not-equal` returns true if the characters `char1` and `char2` are not equal. Both functions ignore case using Unicode rules specified by `style`.

The current implementation only supports one style of comparison:

`:simple-case-fold`

Compares characters using the simple case folding rules in Unicode 6.3.0.

**See also**

`unicode-char-greaterp`
The functions `unicode-char-greaterp` and `unicode-char-lessp` return true if the character `char1` is greater than (or for `unicode-char-lessp`, less than) the character `char2`, similarly to `cl:char-greaterp` and `cl:char-lessp` but ignoring case using Unicode rules specified by `style`.

The current implementation only supports one style of comparison:

`:simple-case-fold`

- Compares characters using the simple lowercase folding rules in Unicode 6.3.0.

See also:

- `unicode-char-equal`
- `unicode-char-not-greaterp`
# Function

**unicode-char-not-greaterp**

**unicode-char-not-lessp**

**Summary**

Compares two characters, ignoring case using specified Unicode rules.

**Package**

lispworks

**Signatures**

- `unicode-char-not-greaterp char1 char2 &key style => flag`
- `unicode-char-not-lessp char1 char2 &key style => flag`

**Arguments**

- `char1` A character
- `char2` A character
- `style` A keyword

**Values**

- `flag` A generalized boolean

**Description**

The functions `unicode-char-not-greaterp` and `unicode-char-not-lessp` return true if the character `char1` is not greater (or for `unicode-char-not-lessp`, not less) than the character `char2`, similarly to `cl:char-not-greaterp` and `cl:char-not-lessp` but ignoring case using Unicode rules specified by `style`.

The current implementation only supports one style of comparison:

- `:simple-case-fold`
  
  Compares characters using the simple lowercase folding rules in Unicode 6.3.0.

**See also**

- `unicode-char-equal`
- `unicode-char-greaterp`
**unicode-lower-case-p**  
*Function*

**Summary**
Returns a value like `cl:lower-case-p`, but using specified Unicode rules.

**Package**
lispworks

**Signature**
`unicode-lower-case-p char &key style => flag`

**Arguments**
- `char` A character
- `style` A keyword

**Values**
- `flag` A generalized boolean

**Description**
The function `unicode-lower-case-p` returns `flag` as true if `char` is lowercase according to the Unicode rules specified by `style`.

The current implementation only supports one style:

`:general-category`

Use the "general category" for `char` in Unicode 6.3.0.

**See also**
- `unicode-both-case-p`
- `unicode-upper-case-p`

---

**unicode-string-equal**

**unicode-string-not-equal**  
*Functions*

**Summary**
Compares two strings, ignoring case using specified Unicode rules.

**Package**
lispworks

**Signatures**

```
unicode-string-equal string1 string2 &key start1 start2 end1 end2 style => flag
```
The functions `unicode-string-equal` and `unicode-string-not-equal` compare the designated substrings of `string1` and `string2`, ignoring case using Unicode rules specified by `style`. The values of `start1` and `start2` default to 0, while the values of `end1` and `end2` default to `nil`.

The returned value `flag` of `unicode-string-equal` is true if the strings are equal and false otherwise.

The returned value `mismatch-index` of `unicode-string-not-equal` is the index where the strings mismatch (as an offset from the beginning of `string1`) or `nil` otherwise.

The current implementation only supports one style of comparison:

: `simple-case-fold`

  Compares each character of the strings using the simple case folding rules in Unicode 6.3.0.

See also `choose-unicode-string-hash-function`
**Functions**

**unicode-string-greaterp**

**unicode-string-lessp**

### Summary

Compares two strings, ignoring case using specified Unicode rules.

### Package

lispworks

### Signatures

**unicode-string-greaterp** string1 string2 &key start1 start2 end1 end2 style => mismatch-index

**unicode-string-lessp** string1 string2 &key start1 start2 end1 end2 style => mismatch-index

### Arguments

- **string1** A string designator
- **string2** A string designator
- **start1, end1** Bounding index designators of string1
- **start2, end2** Bounding index designators of string2
- **style** A keyword

### Values

- **mismatch-index** A bounding index of string1 or nil

### Description

The functions **unicode-string-greaterp** and **unicode-string-lessp** compare the designated substrings of string1 and string2, similarly to cl:string-greaterp and cl:string-greaterp but ignoring case using Unicode rules specified by style. The values of start1 and start2 default to 0, while the values of end1 and end2 default to nil.

The value of mismatch-index is the index where the strings mismatch (as an offset from the beginning of string1) if substring1 is greater (or for **unicode-string-lessp**, less) than substring2, or nil otherwise.

The current implementation only supports one style of comparison:

: simple-case-fold
Compares each character of the string using the simple lowercase folding rules in Unicode 6.3.0.

See also  
- `unicode-string-equal`
- `unicode-string-not-greaterp`

### Functions

**unicode-string-not-greaterp**

**unicode-string-not-lessp**

**Summary**

Compares two strings, ignoring case using specified Unicode rules.

**Package**

`lispworks`

**Signatures**

```lisp
unicode-string-not-greaterp string1 string2 &key start1 start2 end1 end2 style => mismatch-index
unicode-string-not-lessp string1 string2 &key start1 start2 end1 end2 style => mismatch-index
```

**Arguments**

- `string1`  
  A string designator

- `string2`  
  A string designator

- `start1, end1`  
  Bounding index designators of `string1`

- `start2, end2`  
  Bounding index designators of `string2`

- `style`  
  A keyword

**Values**

- `mismatch-index`  
  A bounding index of `string1` or `nil`

**Description**

The functions `unicode-string-not-greaterp` and `unicode-string-not-lessp` compare the designated substrings of `string1` and `string2`, similarly to `cl:string-not-greaterp` and `cl:string-not-lessp` but ignoring case using Unicode rules specified by `style`. The values of `start1` and `start2` default to 0, while the values of `end1` and `end2` default to `nil`.  

1197
The value of `mismatch-index` is the index where the strings mismatch (as an offset from the beginning of `string1`) if `substring1` is not greater (or for `unicode-string-not-lessp`, not less) than `substring2`, or `nil` otherwise.

The current implementation only supports one style of comparison:

```
:simple-case-fold
```

Compares each character of the string using the simple lowercase folding rules in Unicode 6.3.0.

See also

- `unicode-string-equal`
- `unicode-string-greaterp`

---

### `unicode-upper-case-p`

#### Function

**Summary**

Returns a value like `cl:upper-case-p`, but using specified Unicode rules.

**Package**

`lispworks`

**Signature**

`unicode-upper-case-p char &key style => flag`

**Arguments**

- `char` A character
- `style` A keyword

**Values**

- `flag` A generalized boolean

**Description**

The function `unicode-upper-case-p` returns `flag` as true if `char` is uppercase according to the Unicode rules specified by `style`.

The current implementation only supports one style:

```
:general-category
```
Use the "general category" for char in Unicode 6.3.0.

See also
- unicode-both-case-p
- unicode-lower-case-p

**Function**

### user-preference

**Summary**
Gets or sets a persistent value in the user’s registry.

**Package**
lispworks

**Signature**

```lisp
user-preference path value-name &key product => value, valuep
```

**Signature**

```lisp
(setf user-preference) value path value-name &key product => value
```

**Arguments**

- **path**
  A string or a list of strings.
- **value-name**
  A string.
- **product**
  A keyword.

**Values**

- **value**
  A Lisp object.
- **valuep**
  A boolean.

**Description**

The function **user-preference** reads the value of the registry entry **value-name** under **path** under the registry path defined for **product** by (setf product-registry-path). If the registry entry was found a second value \( t \) is returned. If the registry entry was not found, then **value** is **nil**.

The function (setf user-preference) sets the value of that registry entry to **value**.

If **path** is a list of strings, then it is interpreted like the directory component of a pathname. If **path** is a string, then any directory separators should be appropriate for the platform -
that is, use backslash on Windows, and forward slash on non-Windows systems.

Notes

1. When value is a string, user-preference stores a print-escaped string in the registry and reads it back with read-from-string. Therefore it may not work with string values stored by other software.

2. While product can in principle be any Lisp object, values of product are compared by eq, so you should use keywords.


Example

This example is on Microsoft Windows. Note the use of backslashes as directory separators in the path argument:

```
(setf (user-preference "My Stuff\FAQ"
       "Ultimate Answer"
       :product :deep-thought)
  42)
 => 42
```

This is equivalent to the previous example, and is portable because we avoid the explicit directory separators in the path argument:

```
(setf (user-preference (list "My Stuff" "FAQ")
       "Ultimate Answer"
       :product :deep-thought)
  42)
 => 42
```

We can retrieve values on Windows like this:
(user-preference "My Stuff\FAQ"
  "Ultimate Answer"
  :product :deep-thought)
=>
42

We can retrieve values on any platform like this:

(user-preference (list "My Stuff" "FAQ")
  "Ultimate Question"
  :product :deep-thought)
=>
nil
nil

See also
  copy-preferences-from-older-version
  product-registry-path

**when-let**  

*Macro*

**Summary**
Executes a body of code if a form evaluates to non-nil, propagating the result of the form through the body of code.

**Package**  
lispworks

**Signature**
when-let (var form) &body body => result

**Arguments**

- **var**
  A variable whose value is used in the evaluation of body.

- **form**
  A form, which must evaluate to non-nil.

- **body**
  A body of code to be evaluated conditionally on the result of form.

**Values**

- **result**
  The result of evaluating body using the value var.
**when-let**

*Macro*

Summary: Executes a body of code if a series of forms evaluates to non-nil, propagating the results of the forms through the body of code.

Package: lispworks

Signature: when-let* bindings &body body => result

bindings ::= ((var form)*)

Arguments:
- **var**: A variable whose value is used in the evaluation of body.
- **form**: A form, which must evaluate to non-nil.
- **body**: A body of code to be evaluated conditionally on the result of form.

Values:
- **result**: The result of evaluating body using the value var.
Description
The macro `when-let*` expands into nested `when-let` forms. The bindings are evaluated in turn as long as each returns non-nil. If the last binding evaluates to non-nil, `body` is executed. Within the code `body`, each variable `var` is bound to the result of the corresponding form `form`.

Example
(defmacro divisible (n &rest divisors)  
  `(when-let* ,((loop for div in divisors  
                    collect (list (gensym)  
                                   (zerop (mod n div))))  
                 t))

See also `when-let`

whitespace-char-p

Function

Summary
Tests whether a character represents white space.

Package
`lispworks`

Signature
`whitespace-char-p char => result`

Arguments
`char` A character.

Values
`result` A boolean.

Description
The function `whitespace-char-p` is a predicate for whitespace, as described in the standard:

“Space and non-graphic characters that only moved the print position.”

`result` is `t` if `char` represents white space, and `nil` otherwise.

If the value of `*extended-spaces*` is `t`, then U+3000 Ideographic Space is also considered whitespace.

See also `*extended-spaces*`
**with-action-item-error-handling**

*Macro*

**Summary**
Executes a body of code across action lists and items, signaling errors and then continuing to the next action item.

**Package**
lispworks

**Signature**

```
with-action-item-error-handling action-list-var action-item-var ignore-errors-p &body body
```

**Arguments**

- `action-list-var` A variable.
- `action-item-var` A variable.
- `ignore-errors-p` A boolean.
- `body` A body of Lisp code.

**Description**
The macro `with-action-item-error-handling` executes the body with `action-list-var` and `action-item-var` are bound to the action list and item respectively. If `ignore-errors-p` is set to `t` then errors are handled. The behavior of the handler is to signal a warning in which the action-list, item and original error are all reported; execution then continues with the next action-item.

**Example**

```lisp
(defun my-execution-function (the-action-list
  other-args
  &key ignore-errors-p
  &allow-other-keys)
  (with-action-list-mapping (the-action-list
    an-action-item
    action-item-data)
    (with-action-item-error-handling (the-action-list
      an-action-item
      ignore-errors-p)
      (do-something-interesting-first)
      (apply (car action-item-data) other-args (cdr
        action-item-data))))))
```

If this function was invoked with the keyword argument `:ignore-errors-p` `t`, and an error was signaled while exe-
cuting the body-form(s) for one of the action-items, then a warning such as:

```
Warning: Got an error 'The variable *PREV-STATE* is unbound.' while executing action "Initialize State" in list "Startup Inits".
```

would be signaled and execution would continue with the next action-item.

See also  *handle-missing-action-in-action-list*

**with-action-list-mapping**

*Macro*

**Summary**
Maps over an action list's actions with given variables bound to the executing action and its data.

**Package**
lispworks

**Signature**

```
with-action-list-mapping action-list item-var data-var
 &optional post-process &body body)
```

**Arguments**

- **action-list**  An action list.
- **item-var**  A Lisp symbol.
- **data-var**  A Lisp symbol.
- **post-process**  A keyword.
- **body**  A body of Lisp code.

**Description**
The macro **with-action-list-mapping** maps over an action-list's action-items. During execution, the symbols specified for **item-var** and **data-var** are bound to the executing action-item and its data respectively. See **execute-actions** for more on post-processing.

If this function is invoked with the keyword argument **:post-process :collect**, a list the values returned by each action-item's `setf` operation are returned.
Examples

(defun my-execution-function
  (the-action-list other-args
    &key (post-process nil)
    &allow-other-keys)
       (declare (ignore other-args))
  (with-action-list-mapping (the-action-list
    an-action-item
    action-item-data
    post-process)
      (do-something-interesting-first)
      (setf (symbol-value (car action-item-data))
        (apply (cadr action-item-data)
          (cddr action-item-data))))))

See also  execute-actions

with-unique-names  

Macro

Summary  Returns a body of code with each specified name bound to a similar name.

Package  lispworks

Signature  with-unique-names (&rest names) &body body => result

Arguments  names  The names to be rebound in body.

body  The body of code within which names are rebound.

Values  result  The result of evaluating body.

Description  Returns the body with each name bound to a symbol of a similar name (compare gensym).

Example  After defining
(defmacro lister (p q)
  (with-unique-names (x y)
    `(let ((,x (x-function))
          (,y (y-function)))
      (list ,p ,q ,x ,y))))

the form (lister i j) macroexpands to

(LET* ((#:X-88 (X-FUNCTION))
        (#:Y-89 (Y-FUNCTION)))
      (LIST i j #:X-88 #:Y-89))

See also rebinding
This chapter describes symbols available in the LW-JI package, the LispWorks Java interface.

The uses of these symbols are discussed in Chapter 15, “Java interface”.

**call-java-method**

*Function*

**Summary**

Call a Java method.

**Package**

lw-ji

**Signature**

call-java-method method-name &rest args

**Arguments**

- method-name: A string.
- args: Lisp objects.

**Description**

The function call-java-method calls a Java method on the supplied args.

*method-name* must specify the full name of the Java method to call, including the package, class and method name, for
example "java.io.File.exists". call-java-method first uses the string to lookup a caller, and if that fails it produces a caller in the same way that define-java-caller and setup-java-caller do and caches it. It then uses the caller to call the Java method with args, and return the result.

The process of actual calling is the same as in ordinary Java callers defined by define-java-caller. See the entry for define-java-caller for details.

If method-name is incorrect (does not have class and method name, class cannot be found or method cannot be found), call-java-method signals an error of type call-java-method-error, which reports the actual failure.

Notes call-java-method needs to lookup the caller using the string, so the call is slightly slower than calls for ordinary Java callers, but the difference is not significant. There is also no way to verify that the string is correct. It also has to keep some extra code that can be shaken out if only define-java-caller is used, but not much. If you find it convenient, there is no reason not to use it.

See also define-java-caller call-java-method-error “Defining specific callers” on page 202

call-java-method-error

<table>
<thead>
<tr>
<th>Summary</th>
<th>call-java-method failed to find the method.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>lw-ji</td>
</tr>
<tr>
<td>Superclasses</td>
<td>java-interface-error</td>
</tr>
<tr>
<td>Subclasses</td>
<td>None</td>
</tr>
</tbody>
</table>
The condition class `call-java-method-error` is signaled when `call-java-method` failed to find the method.

See also `call-java-method`

**caching-java-exceptions**
**caching-exceptions-bind**

*Macros*

**Summary**
Execute Lisp code with a catch for Java exceptions.

**Package**
lw-ji

**Signature**
catching-java-exceptions &body body

catching-exceptions-bind (result exception) form &body body

**Arguments**
result A variable.
exception A variable.
form A Lisp form.
body Lisp code.

**Description**
The macro `caching-java-exceptions` executes `body` with a catch for Java exceptions. The code of `body` is executed normally, and if no Java exception is signaled through the execution, returns whatever `body` returns. If there is an exception, instead of signaling an error of class `java-exception`, `caching-java-exceptions` returns two values: `nil` and the Java exception object (analogous to `cl:ignore-errors`).

The macro `caching-exceptions-bind` executes the `form` and binds `result` and `exception` to the first two return values if there was no exception. If there was an exception they are bound to `nil` and the exception. It then executes the code of `body` within the scope of the bind. `caching-exceptions-bind` is equivalent to
(multiple-value-bind (result exception)  
  (catching-java-exceptions form)  
  body)

Notes

1. `jobject-string`, `jobject-class-name` and `jobject-of-class-p` are useful general utilities for deciding what to do with the exception. For fine-grained handling, you will need to access the exception using your own callers or `call-java-method` when applicable.

2. These macros have no effect on signaling and handling of other errors in Lisp, except that they prevent Java exceptions from being signaled as errors.

3. Some exceptions can happen during normal execution and handled by the system in a user-invisible way (analogous to the way that `try` in Java code does). These macros do not affect the behavior for these cases, so even though when running under a Java debugger you may see an exception, it will not necessarily be visible with these macros.

4. In general, these macros are less useful in high-level code, because they cause exceptions to throw out, preventing them from being signaled as Lisp errors and handled by error handler in the scope of `body` (for `catching-java-exceptions`) or `form` (for `catching-exceptions-bind`). They should normally be used in low-level code that actually does Java calls, with any Lisp error handlers wrapped around them.

5. For simple handling of exceptions you can use standard handlers (`cl:handler-case`, `cl:handler-bind`), for `java-exception` and its subclasses.

See also `jobject-string`  
`jobject-class-name`  
`jobject-of-class-p`
check-lisp-calls-initialized  

Function

Summary  Tests whether calls from Java into Lisp can work.

Package  lw-ji

Signature  check-lisp-calls-initialized => result

Arguments  None.

Values  result  A boolean.

Description  The function check-lisp-calls-initialized returns true if Lisp calls have been initialized successfully, which means that calls from Java into Lisp can work.

The main factor that may affect successful initialization of Lisp calls is the availability of the class com.lisp-works.LispCalls, which comes from LispWorks and will not be available if you do not make it available to the Java Virtual Machine in some way.

Notes  On Android the LispCalls class is available because it is in the lispworks.jar file that must be included in the application.

See also  “Calling from Java to Lisp” on page 206

create-instance-from-jobject  

Function

Summary  Create a CLOS instance based on a jobject.

Package  lw-ji

Signature  create-instance-from-jobject jobject &optional errorp => instance
The LW-JI Package

Arguments

- **object**: A `jobject`.
- **errorp**: A generalized boolean.

Values

- **instance**: A CLOS object.

Description

The function `create-instance-from-jobject` creates a CLOS instance based on the `jobject` object.

`jobject` must be a `jobject`. Its class name (that is, the result of `jobject-class-name`) must have been associated with the name of a CLOS subclass of `standard-java-object` using `record-java-class-lisp-symbol` (the importing interface, when defining a class, does it automatically).

`create-instance-from-jobject` uses the record to find the class, and makes the CLOS instance by calling `make-instance`, passing it `jobject`.

The result is an instance of the CLOS class, which can be passed to Java interface functions and Java methods.

If `create-instance-from-jobject` fails to find the CLOS class it signals an error if `errorp` is non-nil, otherwise it returns `nil`. The default value of `errorp` is true.

See also

- `record-java-class-lisp-symbol`
- “CLOS partial integration” on page 218

---

**create-instance-object-list**

**create-instance-object**

*Functions*

Summary

Construct a `jobject` for a CLOS instance.

Package

lw-ji

Signature

- `create-instance-object-list instance args => jobject`
- `create-instance-object instance &rest args`
Arguments

instance An instance of a subclass of standard-java-object.

args A list or t.

Values

jobject A jobject.

Description

The functions create-instance-jobject-list and create-instance-jobject construct a jobject for the CLOS instance instance and set its slot to that jobject.

The type of instance (that is, the class name of its class) must have been associated with a Java constructor by passing it as the class-symbol argument to define-java-constructor or setup-java-constructor.

create-instance-jobject just calls create-instance-jobject-list with instance and args.

create-instance-jobject-list constructs the jobject using the args. args can be either the list of arguments for the constructor (the list may be nil), or t, in which case create-instance-jobject-list uses default-constructor-arguments to create a list of arguments and uses it instead.

The Java constructor is called in the same way that it would be called by the caller that is defined by define-java-constructor. See define-java-constructor for details of calling. The result of the construction is stored in instance and is returned.

Notes

1. The importing interface, when defining a class, automatically generates the define-java-constructor form passing it the class-symbol argument. define-java-constructor also defines a caller for the constructor, which can be used independently. create-instance-jobject-list and create-instance-jobject do not actually call it, but share information with it.
2. `create-instance-object-list` and `create-instance-object` ignore the current `jobject` in `instance`, if there is one. There is no problem calling `create-instance-object-list` and `create-instance-object` repeatedly on the same instance.

See also `define-java-constructor`

**create-java-object**

*Function*

**Summary**  Invoke the Java constructor.

**Package**  *lw-ji*

**Signature**  
`create-java-object class-name &rest args => result`

**Arguments**

- `class-name`  A string.
- `args`  Lisp objects.

**Values**

- `result`  A Java object.

**Description**  The function `create-java-object` invokes the Java constructor for the class `class-name` with the supplied `args`.

`class-name` must specify a Java class. `create-java-object` first uses `class-name` to lookup a caller, and if that fails it produces a caller in the same way that `define-java-constructor` and `setup-java-constructor` do and caches it. It then uses the caller to call the constructor with `args`, and returns the result.

The process of actual calling is the same as in ordinary Java callers defined by `define-java-constructor`. See the entry for `define-java-constructor` for details.

If the string is incorrect (that is, it does not look like a Java class name or the class cannot be found), `create-java-object`
object signals an error of type create-java-object-error, which reports the actual failure.

Notes create-java-object needs to lookup the caller using the string, so the call is slightly slower than calls for ordinary Java constructors, but the different is not significant. It also has to keep some extra code that can be shaken out if only define-java-constructor is used, but not much. If you find it convenient, there is no reason not to use it.

See also define-java-constructor

create-java-object-error  
Condition  
Summary create-java-object failed to find constructors.

Package lw-ji

Superclasses java-interface-error

Subclasses None.

Description The condition class create-java-object-error is signaled when create-java-object failed to find constructors.

See also create-java-object

default-constructor-arguments  
Generic Function  
Summary Returns a default list of arguments to pass to the constructor.

Package lw-ji

Signature default-constructor-arguments instance => list
**default-constructor-arguments**

**Method signatures**

```
default-constructor-arguments (instance standard-java-object) => nil
```

**Arguments**

`instance` An instance of a subclass of `standard-java-object`.

**Values**

`list` A list.

**Description**

The generic function `default-constructor-arguments` returns a default list of arguments to pass to the constructor.

It is used by `create-instance-jobject-list` when its `args` argument is `t`. `default-constructor-arguments` is also used by the `cl:initialize-instance` method of `standard-java-object` when `:construct` is passed with value `t`. It is intended for you to specialize on your own classes.

The default method returns `nil`, which is sometimes useful, but in most cases you probably need to pass something.

**See also**

`create-instance-jobject-list`

---

**default-name-constructor**

**Function**

**Summary**

The default `name-constructor` used by the importing interface.

**Package**

`lw-ji`

**Signature**

```
default-name-constructor prefix method-or-field-name => symbol-name
```

**Arguments**

`prefix` A string.

`method-or-field-name` A string.

**Values**

`symbol-name` A string.
The function `default-name-constructor` is the default `name-constructor` used by the importing interface. See `generate-java-class-definitions` for a description of what it does.

**define-field-accessor**

*Macro*

**Summary**

Defines a Java field accessor.

**Package**

`lw-ji`

**Signature**

```
define-field-accessor name class-name field-name static-p &optional is-final => name
```

**Arguments**

- `name` A symbol.
- `class-name` A string.
- `field-name` A string.
- `static-p` A boolean.
- `is-final` A boolean.

**Values**

- `name` A symbol.

**Description**

The macro `define-field-accessor` defines a field accessor for a field in a Java class. The arguments are interpreted as by `setup-field-accessor`.

Unlike `setup-field-accessor`, `define-field-accessor` does not look up anything. The accessor does the look up first time it is called, and signals an error if something failed. This error should be of type:

```
java-class-error
Failed to find the class.
```

```
java-field-error
```

1219
Failed to find the field, or found the field but wrong static-p value.

**field-access-exception**

Got an exception trying to access the field.

**Notes**

1. In general, accessing fields should be avoided, because they are typically a less well-defined and implemented interface than methods, but sometimes it is necessary.

2. The importing interface generates appropriate **define-field-accessor** forms for public fields.

**See also** **setup-field-accessor**

---

**define-java-caller**

**define-java-constructor**

**Macros**

**Summary**

Define a Java caller, which is a function that calls a Java method or a constructor.

**Package**

lw-ji

**Signature**

```
(define-java-caller name class-name method-name &key signatures => result)
```

**Signature**

```
(define-java-constructor name class-name &key class-symbol signatures => result)
```

**Arguments**

- **name**  A symbol.
- **class-name**  A string.
- **method-name**  A string.
- **signatures**  A list of strings.
- **class-symbol**  A symbol.

**Values**

- **result**  name or nil.
The macros **define-java-caller** and **define-java-constructor** define a Java caller, which is a function that calls a Java method or a constructor. Once this the caller is defined, calls to *name* ultimately invoke the Java method or constructor.

*class-name* must be the full name of a Java class, in the correct case. The `.` in the name may be replaced by `'/`.

*method-name* must be a public method name of the class, with the correct case.

*class-symbol*, when it is non-nil, must name a class. It creates a mapping from the class to the constructor info, which allow functions like **make-java-instance** and **create-instance-jobject** to construct a *jobject* for an instance of the class named by *class-symbol*.

The effect of these macros is to set the symbol function of *name* to a function that calls the method in the class or the constructor of the class. When there is more than one method with the same name or more than one constructor (that is, it is overloaded), the function decides dynamically which of these to call, based on the arguments it gets.

For a successful call to *name*, it needs to be called with the correct arguments for the Java method. For an ordinary method, this must include the object on which the method should be applied, followed by the arguments of the method. For static methods and constructors, the arguments to *name* are just the arguments to the method/constructor.

For arguments of primitive type or a matching Java class (for example **Integer**), the Lisp argument must be either a Lisp object of matching type (see “Types and conversion between Lisp and Java” on page 198), or a *jobject* of the corresponding Java class. For strings (that is argument type **java.lang.String**) the argument must be a string, **nil**, or a *jobject* of type **java.lang.String**. For other non-primitive
types, the argument must be a \texttt{object} of the correct class or \texttt{nil}. \texttt{nil} is passed as Java \texttt{null} for non-primitive types.

If the return value type is a primitive type or \texttt{String}, the function converts the result of the method to the matching Lisp type before returning it. For other return types, the function returns a \texttt{object} representing the Java object, or \texttt{nil} if the method returned \texttt{null}. Constructors always return \texttt{objects}.

When the method is an ordinary method (not static and not constructor), the invocation is virtual (normal Java behavior), which means that if the object is of a subclass of the \texttt{class-name} argument, it may invoke a method that is defined in a subclass of \texttt{class-name}.

Unlike the functions \texttt{setup-java-caller} and \texttt{setup-java-constructor}, the macros \texttt{define-java-caller} and \texttt{define-java-constructor} do not do any actual lookup, they just set up the symbol function and therefore they do not require running Java to perform the definition. They are also recognized by the Editor as definer forms, so source finders like the Editor command \texttt{Find Source} can locate them. These macros are intended as the main method of defining callers. They are produced by the importing interface to actually define the callers.

For callers defined by these macros, the actual lookup happens the first time the function is invoked, or for \texttt{define-java-caller} by \texttt{verify-java-caller} or \texttt{verify-java-callers}. If the lookup fails during the function call, an error is signaled of type \texttt{java-class-error} (when the class cannot be found) or \texttt{java-method-error} (when the method cannot be found).

The macros (when successful) return \texttt{name}. 
Notes

1. There is no difference in performance between functions defined by these macros and functions defined by `setup-java-caller` and `setup-java-constructor`. If you use `setup-java-caller` and `setup-java-constructor` in a delivered application then extra machinery is retained.

2. If you need several `define-java-caller` forms with the same class, consider using `define-java-callers`.

3. If you need many `define-java-caller` forms with the same class, you may want to use the importing interface. Even if you want to define your own names for the callers, you can either pass `name-constructor` to the import function, or use `write-java-classDefinitions-to-file` and edit the definitions that it generated (which saves typing the method names).

4. For methods it is possible to use `verify-java-callers` or `verify-java-caller` at run time to check that the methods are found, which is a way of guarding against typing errors in entering the method name.

5. There is no restriction on defining more than one caller for the same method or constructor.

6. Unlike `setup-java-caller` and `setup-java-constructor`, the `name` argument is not evaluated.

See also
- `setup-java-caller`
- `define-java-callers`
- `write-java-classDefinitions-to-file`
- `import-java-classDefinitions`
- `verify-java-callers`
- `verify-java-caller`
- “Defining specific callers” on page 202

**define-java-callers**

*Macro*

**Summary**
Define multiple Java callers for methods in the same class.
The macro `define-java-callers` defines multiple Java callers for methods in the same class.

`class-name` must specify a Java class by its full name.

Each item of `method-specs` must be a list where the first element is a symbol (the Java caller name), the second element is a string (the method name) and optionally followed by keyword/value pairs for `define-java-caller`. `define-java-callers` processes each item by inserting the `class-name` after the Java caller name, and then using the result as the arguments to `define-java-caller`:

```lisp
(define-java-callers class-name
  (caller-name1 method-name1)
  (caller-name2 method-name2))
=>
(progn
  (define-java-caller caller-name1 class-name method-name1)
  (define-java-caller caller-name2 class-name method-name2)
  class-name)
```

`define-java-callers` is a more compact way to write several methods for the same class, but functionally it is identical to using `define-java-caller` explicitly.

`define-java-callers` returns the `class-name`.

See also `define-java-caller`
define-lisp-proxy

Summary
Defines a Lisp proxy.

Package
lw-ji

Signature
define-lisp-proxy name &body interface-and-method-descs => name

Arguments
name A non-nil symbol.
interface-and-method-descs A body of Lisp code.

Values
name A non-nil symbol.

Description
The macro define-lisp-proxy defines a Lisp proxy, which means creating a Lisp proxy definition and attaching it to name, which can then be used to create Lisp proxies, which are Java proxies where methods invocation ends up calling Lisp functions.

define-lisp-proxy parses interface-and-method-descs to a proxy definition, and attaches it to name. This operation is a "load-time" operation: it does not require running Java, and does not create any proxy. The name can then be used at run time as argument to make-lisp-proxy or make-lisp-proxy-with-overrides, or to the Java method com.lisp-works.LispCalls.createLispProxy. The result of any these calls is a proxy that implements the interfaces listed in interface-and-method-descs, and can be used in Java whenever an object that implements any of these interfaces is required.

interface-and-method-descs describes the Java interfaces to implement and the Lisp functions to call. It is parsed as a body of Lisp forms.

Each element in the list must be either a string which is the Java interface name, or a list where the cl:car is the Java interface name. Each item specifies a Java interface to imple-
ment, except that one item (at most) may specify options relating to the whole proxy definition, by using a list starting with the keyword `:options` instead of giving an interface name.

When the item is a list starting with an interface name, the rest of the list are method specifications. Note that you do not need to have a method specification for each method of the interface.

Each method specification must be a list, where the first element is a string with the name of the Java method, and the second element is the symbol specifying what Lisp function to call for this method. The symbol specifies the function to call except when it is overridden (see below about "Overriding"). In some cases, you will want to always override the function to call (typically when you want to use a closure as the function), in which case the symbol can be and should be a keyword (which is ignored by the verifying functions), but does not have to be. See below for how the calling of the Lisp function is done.

The rest of the method specification can contain keyword/value pairs. Currently, the only supported keyword is `:with-user-data`, which takes a boolean value, overrides the default value of `:with-user-data` of the proxy definition. The default of `:with-user-data` of the definition defaults to `nil`, and can be changed in the `:options`. The value of `:with-user-data` specifies whether to pass the user-data of a proxy to the Lisp function.

The `:options` item is specified by an item in `interface-and-method-descs` where the `cl:car` is the keyword `:options`. The rest of the item is keyword/value pairs. The keywords currently supported are:

`:default-function`

Specifies the default function to call for methods which do not have a Lisp function. This function is applied to the arguments of
the method preceded by the method-name, and if :default-function-with-user-data is non-nil also with user-data preceding the method-name.

The default function can be overridden by make-lisp-proxy and make-lisp-proxy-with-overrides.

:default-function-with-user-data
A boolean specifying whether the user-data of a proxy should be passed when the default function is called. When it is non-nil, the user-data is passed as the first argument to the default-function (or the function that overrides it). The default value of :default-function-with-user-data is nil.

:with-user-data
A boolean specifying whether the default for calling functions in the proxy definition is with user-data or not. Each method description can override it as described above. The default value of :with-user-data is nil.

:print-name
Must be a string or a symbol. Specifies the first part of the print-name of each proxy.

:jobject-scope
One of :global, :local or nil. This controls the scope of object arguments (that is, arguments that are not of primitive type or string). With the default value :global, objects are passed as global objects and can be used indefinitely. When :jobject-scope is :local, objects are passed as a local object, which means that they must
not be used outside the scope of the function that is invoked by the proxy. Using a local \texttt{jobject} out of scope can cause the system to crash (rather than call \texttt{cl: error}). When \texttt{:jobject-scope} is \texttt{nil}, \texttt{jobjects} are not passed at all to the functions. Note that means that the number of arguments that the functions in the proxy receive is different when \texttt{:jobject-scope} is \texttt{nil}, because only arguments of primitive type or strings are passed.

If you use \texttt{:jobject-scope} \texttt{:local}, the function can convert it to global using \texttt{job-\textit{object-ensure-global}}, and then it can be used out of scope.

The default value of \texttt{:jobject-scope} is \texttt{:global}.

\textbf{user-data}

The \textit{user-data} is set up for each individual proxy object by \texttt{make-lisp-proxy} or \texttt{make-lisp-proxy-with-overrides}, and thus allows you to associate each individual proxy with an arbitrary Lisp object. The proxy definition determines whether to use it when calling the Lisp functions in the proxy definition. The default value of \textit{user-data} is \texttt{nil}, so if you want to use it you need to specify it by using \texttt{:with-user-data}, either in the \texttt{:options} which would give the default value for all calls in the definition, or in individual method specifications. When \textit{user-data} is passed, it is always passed to the Lisp function as the first argument. Another way to individualize proxies is to use overriding, which also allows you to use closures.
Overriding

When `make-lisp-proxy` or `make-lisp-proxy-with-overrides` make a proxy, they can specify overriding of some of the symbols in the proxy definition. Overriding here means mapping one symbol to another symbol or a function object. When a symbol is supposed to be called and it is overridden, the target of the mapping is called rather than the symbol. Note that the overriding is specific to each individual proxy rather to the proxy definition, and therefore you can have different proxies using the same proxy definition (and hence implementing the same interface(s)), but calling different Lisp functions. An advantage of overriding is that it allows you to use closures created at run time instead of symbols.

See the entry for `make-lisp-proxy` for how the overriding is created.

Calling the Lisp function

After a proxy is created from a proxy definition, any invocation of a Java method on it (except the Object methods `toString`, `equals` and `hashCode`) enters Lisp.

When a method is invoked on a proxy (normally from Java, but can be done from Lisp too), the steps for invoking your Lisp function are:

1. Check whether the item in the `interface-and-method-descs` for the interface of the method contains a method specification with the `method-name` of the method.

2. Convert the Java method arguments to Lisp arguments where possible. See “Types and conversion between Lisp and Java” on page 198.

   Note that that if the first step above found a method specification, and it contains the keyword `:jobject-scope`, it affects the way non-primitive arguments are processed as described above.
3. Calling the user code:

(i) If a method specification was found in the first step above:

a) Take the symbol from the method specification, then

b) Check whether the symbol is overridden, and if it is use the target as a function to call. Otherwise, check whether the symbol is fbound, and if it is use it as a function to call, then

c) If as a result of (b) there is a function to call, check whether it should be called with the \textit{user-data}. If \texttt{with-user-data} was used in the method specification then use its value, otherwise if \texttt{with-user-data} was used in \texttt{options} item use this value, otherwise default to \texttt{nil}, then

d) Apply the function: if using \textit{user-data}, apply the function to the \textit{user-data} followed by the Lisp arguments, otherwise apply the function to the Lisp arguments only.

(ii) If the method-specific call in (i) did not happen (no method specification found, or the symbol is not fbound and not overridden), try to apply the default function:

a) If there is a default function, check whether it is overridden and if so use the target as the function to call. Otherwise use the \texttt{default-function} itself as the function to call, then

b) Check whether need to pass \textit{user-data}, which is specified by the \texttt{default-function-with-user-data} in the \texttt{options} item, then

c) Apply the function: if \textit{user-data} needs to be used, apply the function to the \textit{user-data}, \textit{method-name} and the Lisp arguments. Otherwise apply the function to the \textit{method-name} and Lisp arguments.

(iii) If the calls in (i) and (ii) did not happen, an error is signaled. See handling of errors below.
4. Return a value: currently, if the user function returned a Java object (a `jobject` or an instance of `standard-java-object`), it is returned without checking. Otherwise, try to convert it to the appropriate Java object and return it. Otherwise, report it by calling the `java-to-lisp-debugger-hook` (see `init-java-interface`) with a `cl:simple-error` condition and return a default value from the Java method invocation, which is 0 for primitive types or `null` for other types.

**Throwing out and error handling**

The call to the Lisp function is wrapped dynamically such that any throw from it is blocked, and the default value as in the last step above is returned.

In addition, there is a debugger wrapper (using `with-debugger-wrapper`) which calls the `java-to-lisp-debugger-hook` (see `init-java-interface`) with the condition and then calls `cl:abort`. If this abort is not caught by your `cl:abort` restart, it is handled by the "throwing blocker" from the previous paragraph, that is the Java method returns 0 or `null`.

**Verification**

The verification functions `verify-lisp-proxies` and `verify-lisp-proxy` are provided to allow you to do some checking of the correctness of your proxy definition. Two things can be verified:

- That the symbols to be called in the proxy definition are `fbound`. In principle this check could happen at load-time, but that would enforce defining the functions before the proxy. The verification allows you to define the proxies and functions in any order, and then verify all the definitions, for example just before delivery.
• Check that all methods that are declared in the Java interfaces have a method-desc, and report those that do not. This requires running Java.

Note that neither of these issues is actually an error, because the default function can handle them. However it is useful to check them in case you did miss something.

Performance issues

There is a little overhead associated with using setup-lisp-proxy as opposed to define-lisp-proxy, both in the size of the delivered application (very small) and in run time, but the difference is not large enough to prevent using setup-lisp-proxy when it is appropriate.

There is an overhead associated with initializing a proxy definition. It is therefore a bad idea to use setup-lisp-proxy many times.

Overrides and using multiple interfaces add a negligible overhead.

:job-ject-scope with nil or :local are useful optimizations. In proxies that are invoked infrequently, say less than 10 times each second, the difference is probably insignificant, but it is useful for proxies that are called repeatedly by Java code. For example, if you implement the interface "java.io.FilenameFilter" to pass to "java.io.File.list" on large directories, using :job-ject-scope :local or nil will reduce the overhead significantly.

Examples

(example-edit-file "android/android-othello-user")

See also

make-lisp-proxy
make-lisp-proxy-with-overrides
verify-lisp-proxy
verify-lisp-proxies
check-lisp-calls-initialized
**ensure-lisp-classes-from-tree**

*Function*

**Summary**

Creates a Lisp class, and potentially some or all the super-classes as needed based on the tree.

**Package**

lw-ji

**Signature**

```
ensure-lisp-classes-from-tree lisp-name java-class-tree force-p => class
```

**Arguments**

- *lisp-name* 
  A symbol.
- *java-class-tree* 
  A tree.
- *force-p* 
  A generalized boolean.

**Values**

- *class* 
  A class metaobject.

**Description**

The function `ensure-lisp-classes-from-tree` creates a class for `lisp-name`, and potentially some or all the super-classes as needed based on the tree. Note that all references to "class" here are to Lisp classes. `ensure-lisp-classes-from-tree` does not actually know anything about Java.

`ensure-lisp-classes-from-tree` appears the output of the importing interface functions, where it is called with the output of `get-superclass-and-interfaces-tree`. Users can use it as well, but normally using plain `defclass` is much more appropriate.

`java-class-tree` is a tree representing the hierarchy of the Java classes. The structure of the tree is describe in the entry for `get-superclass-and-interfaces-tree`. In general it is assumed that this tree was generated by `get-superclass-and-interfaces-tree`, but you can generate it yourself if
you find it useful, but normally simply using `defclass` to define the classes you want is better.

`force-p` controls whether to force classes to exist or not.

The processing of a node in the tree when `force-p` is `nil` is as follows (note that the `java-class-tree` argument is the first node):

1. Find the symbol corresponding to the class. For the first node, this is the `lisp-name` argument. For other nodes, it first checks whether `record-java-class-lisp-symbol` recorded the `java-class-name` to `lisp-name` mapping, and use it if it did. If not, `ensure-lisp-classes-from-tree` skips this node and use instead the superclass node.

2. Once the symbol is found, `ensure-lisp-classes-from-tree` processes the nodes of the superclass and the nodes of the interfaces, each one of which returns a class, and construct the superclasses list from the result. It remove duplicates from the list, which can happen because interfaces can be implemented by more than one route.

3. Once it got the superclasses, except for the first node, `ensure-lisp-classes-from-tree` checks whether the symbol has got a class definition, and if this class definition inherit from all the superclasses. If it does, it returns this class as the result. If a class is found but is not inheriting all the superclasses, `ensure-lisp-classes-from-tree` redefine it to inherit all the superclasses (ignoring the existing definition), and return it. If the class is not found, `ensure-lisp-classes-from-tree` skips this node and use instead the superclass node.

   For the first node, `ensure-lisp-classes-from-tree` always creates the class.

If `force-p` is true, then `ensure-lisp-classes-from-tree` never fails for any node. Instead, in step 1) when it does not find the symbol it generates a symbol in the same way that
generate-java-class-definitions does by default, and in step 3.) if there is no class it creates it.

Notes

1. ensure-lisp-classes-from-tree does not need running Java.

2. The main purpose of ensure-lisp-classes-from-tree is to create the needed class(es) at load-time without a need for running Java. It is not intended to be used at run time.

3. ensure-lisp-classes-from-tree uses clos:ensure-class to create or redefine classes, so requires keeping CLOS in a delivered image (as described in the LispWorks Delivery User Guide).

4. When the java-class-tree argument matches the Java hierarchy, as it is when it is the result of get-superclass-and-interfaces-tree, if force-p is true ensure-lisp-classes-from-tree generates a full hierarchy with a CLOS class matching each Java class. with force-p nil, at least standard-java-object will always be in the hierarchy, plus any classes that were define by the importing interface or recorded by the user using record-java-class-lisp-symbol.

See also get-superclass-and-interfaces-tree
generate-java-class-definitions

ensure-supers-contain-java.lang.object Function

Summary Checks that at least one of the supplied symbols names a subclass of standard-java-object.

Package lw-ji

Signature ensure-supers-contain-java.lang.object super-symbols lisp-name => nil
The LW-JI Package

Arguments  

- **super-symbols**: A list of symbols.
- **lisp-name**: A symbol.

Values  

- **ensure-supers-contain-java.lang.object** returns nil.

Description  

The function **ensure-supers-contain-java.lang.object** checks that at least one of the symbols in `super-symbols` names a subclass of `standard-java-object` (or `standard-java-object` itself), otherwise it signals an error reporting that the superclasses for `lisp-name` do not have a subclass of `standard-java-object`.

**intern-and-export-list** is a utility function that is used by the importing interface when `lisp-supers` is passed to ensure at load-time that the supers contain a subclass of `standard-java-object`.

See also **generate-java-class-definitions**

---

**field-access-exception**  
**Condition**

Summary  

Conditions signaled when accessing a field gets an exception.

Package  

**lw-ji**

Superclasses  

**field-exception**

Subclasses  

None.

Readers  

**field-access-exception-set-p**

Description  

The condition class **field-access-exception** is a subclass of **field-exception**.

**field-access-exception** is signaled when an attempt to access a field gets an exception. This can occur for various
reasons, for example the new value that was passed for setting is not an acceptable value.

The reader `field-access-exception-set-p` indicates whether the attempted access was setting or reading.

Notes

You can use the `field-exception` readers `field-exception-class-name` and `field-exception-field-name` on conditions of class `field-access-exception`.

See also

`field-exception`

---

**field-exception**

*Condition*

Summary

An abstract class, meaning that it is not signaled. Its readers can be used to access the subclasses.

Package

`lw-ji`

Superclasses

`java-normal-exception`

Subclasses

`field-access-exception`

Readers

`field-exception-class-name`
`field-exception-field-name`

Description

The condition class `field-exception` is a subclass of `java-normal-exception`. `field-exception` is an abstract class, meaning that it is not signaled. Its subclass `field-access-exception` is signaled however and its readers can be used to access those conditions.

`field-exception-class-name` returns the class name and `field-exception-field-name` returns the field name.

See also

`field-access-exception`
**find-java-class**

*Function*

**Summary**

Finds a Java class and returns a `jobject` representing it.

**Package**

lw-ji

**Signature**

`find-java-class class-sym-or-string &optional errorp => result`

**Arguments**

*class-sym-or-string*

A symbol or a string.

**Values**

*result*

A `jobject` or `nil`.

**Description**

The function `find-java-class` finds a Java class and returns a `jobject` representing it.

If `class-sym-or-string` is a string, it should be the full name of class. `find-java-class` allows the '.' in the names to be replaced by '/' (which is how the class is actually looked up). `find-java-class` also recognizes class names of primitives (for example, "int"), and can also find classes for arrays, using the internal syntax with leading '[' character(s).

If `class-sym-or-string` is a symbol, it can be a keyword specifying a primitive class (see the table in “Types and conversion between Lisp and Java” on page 198), one of `:object` or `t` to specify `java.lang.Object`, `:string` to specify `java.lang.String`, or a symbol which is set to a string, in which case the value is used to search for a class.

If `find-java-class` finds the Java class, it returns a `jobject` representing it.

Otherwise, if `errorp` is non-nil it signals an error, otherwise it returns `nil`. The default value of `errorp` is `t`.

**Notes**

For most of the Java interface, you do not actually need to find the class.
**format-to-java-host**

*Function*

Summary  Formats a string and sends it to the Java host.

Package  lw-ji

Signature  

```
format-to-java-host format-string args => result
```

Arguments  

- `format-string`: A format control string.
- `args`: Arguments for `format-string`.

Values  

- `result`: A boolean.

Description  The function `format-to-java-host` sends a message to the Java host.

It creates a message by applying `cl:format` with `destination nil` to `format-string` and `args`, and sends it using `send-message-to-java-host` with `where-keyword :append`.

`result` is the value returned by `send-message-to-java-host`.

See also  `send-message-to-java-host`

**generate-java-class-definitions**

*Function*

Summary  Returns a list of forms which are definitions of Java callers that call the public methods (including constructors) of the supplied class, and accessors for public fields.

Package  lw-ji

Signature  

```
generate-java-class-definitions java-class-name &key lisp-name package-name prefix name-constructor export-p create-defpackage lisp-class-p lisp-supers => list-of-definitions, lisp-name-symbol, package-name-string
```
Arguments

- **java-class-name** A string.
- **lisp-name** A symbol.
- **package-name** A string.
- **prefix** A string or `nil`.
- **name-constructor** A function designator.
- **export-p** A generalized boolean.
- **create-defpackage** A generalized boolean.
- **lisp-class-p** A generalized boolean.
- **lisp-supers** A list of symbols.

Values

- **list-of-definitions** A list.
- **lisp-name-symbol** A symbol.
- **package-name-string** A package name.

Description

The function `generate-java-class-definitions` returns a list of forms which are definitions of Java callers that call the public methods (including constructors) of the class specified by `java-class-name`, and accessors for public fields. These include inherited methods and fields.

`generate-java-class-definitions` is normally used indirectly by using `import-java-class-definitions`, but can also be used directly. `write-java-class-definitions-to-file` and `write-java-class-definitions-to-stream` do the same processing as `generate-java-class-definitions`, and then generate output based on the result.

`java-class-name` must name a Java class, and it must be the precise full name, for example "java.io.File", "android.view.View".

If `lisp-name` is supplied it must be a Lisp symbol. In this case it specifies the package to intern the names of definitions in,
and if a CLOS class is defined, the name of this class. It is also automatically defined as a constant with a value the \textit{java-class-name}. \textit{lisp-name} can also be \texttt{nil}.

If \textit{lisp-name} is not supplied, the system creates a Lisp symbol based on the \textit{java-class-name}. Note that this is different from passing \texttt{nil}, because in the latter case \textit{lisp-name} stays \texttt{nil}.

\textit{package-name} is used only if \textit{lisp-name} is supplied as \texttt{nil}, to specify the package where the names of the definitions are interned. It must be a string containing the package name (in the desired case). The package is created if it does not exist already. If \textit{lisp-name} is \texttt{nil} and \textit{package-name} is \texttt{nil} or not supplied, the current package is used.

\textit{prefix}, if supplied, specifies a prefix to use for the names of the definitions. If \textit{prefix} is not supplied or is \texttt{nil}, the name of the Java class without the package part is uppercased and used as the \textit{prefix} (for example for "java.io.File" the \textit{prefix} is "FILE"). The \textit{prefix} is passed to the \textit{name-constructor} to construct names for the Java callers.

If \textit{name-constructor} is supplied, it must be a function taking two string arguments: the \textit{prefix} and the name of the Java method or field that the Java caller is going to call or access (for constructors, the string "new" is passed as the method name). It must return a string which is then interned (without changing the case) in the package to create the symbol that is used as the name of the caller. The \textit{name-constructor} defaults to a function (\texttt{default-name-constructor}) that concatenates the \textit{prefix} as it is, a dot and uppercase of the method/field name. For example, for the method "exists" in the Java class "java.io.File", the default name constructor with the default \textit{prefix} would generate "FILE.EXISTS".

\textit{export-p} controls whether all the Java callers are exported from the package. If it is \texttt{t} all the Java callers are exported, otherwise they are not. The default of \textit{export-p} is \texttt{t}.
create-defpackage controls what form to generate to do the package manipulation. With the default, generate-java-class-definitions generates a form that check that the package exists, otherwise creates it, and if export-p is t, a form that exports all the symbols. If create-defpackage is non-nil, generate-java-class-definitions generates a defpackage form instead. The default value of create-defpackage is nil.

Note: the reason the create-defpackage defaults to nil is that the defpackage form would contain only the symbols that were defined by the importing, which would be wrong if the package needs to export other symbols too, which is quite likely with the default settings (because other classes in the same Java package will default to use the same Lisp package). create-defpackage is useful when you want to create a package that exports only the definitions for a single Java class.

lisp-supers and lisp-class-p control whether a CLOS class is defined for the Java class. By default, no CLOS class is defined. See in “Creating CLOS class” on page 1244.

The generation of the Java callers and accessors by generate-java-class-definitions is as follows:

1. Based on the arguments as described above, it determines the what package, prefix and name-constructor to use, and whether it has a lisp-name and needs to define a CLOS class. It then finds the definition of the Java class.

2. It uses Java methods to find the names of all the public methods, constructors and fields of the class (including inherited ordinary methods and fields).

3. For each name, it calls the name-constructor with the prefix and the name to generate a symbol name which is then interned in the package to generate a symbol. It then generates a form where the operator is one of the macros.
define-java-caller, define-java-constructor, or define-field-accessor, as appropriate, using the symbol as the name.

For example, with the defaults generating for "java.io.File", the constructor would be defined by

(define-java-constructor FILE.NEW "java/io/File")

The caller for the method "exists" would be defined by:

(define-java-caller FILE.EXISTS "java/io/File" "exists")

the accessor for the field "separator" would be defined by:

(define-field-accessor file.separator "java/io/File" "separator" t t)

Note that generate-java-class-definitions uses '/' rather than '.' as separator between the components. The definer macros accept both '/' and '.' as separators. The actual generated forms may contain additional keywords. Currently methods and constructors have the signatures.

4. generate-java-class-definitions also identifies pairs of methods where one has the name set<something> and the other has the name get<something> or is<something>, which are assumed to be setter and getter for the same field. It then generates a cl:seif definition to allow using cl:seif on the symbol corresponding to get<something> or is<something> name to call the set<something> method.

5. For fields, generate-java-class-definitions also generates a symbol macro with a name that is the symbol name preceded and followed by the * character, and allows getting and the setting the field using this symbol.
The first return value of `generate-java-class-definitions` is a list of forms. The list contains the following forms in this order:

- Package manipulation forms (ensuring the package exist and exporting if required) either as `defpackage` forms or forms that explicitly ensure the existence of the package and do any exporting.

- If there is a `lisp-name`, define it as constant with the class name as value, and record the relation between the Java class name and the symbol (this is used by `create-instance-from-jobject`.

- All the Java callers and accessors that were defined as described in the previous paragraph.

- If a CLOS class is needed, a form to create the class.

`generate-java-class-definitions` also returns the `lisp-name` (supplied or generated) as second return value, and the package name of the package that it used as third value.

**Creating CLOS class**

**Note:** see the discussion “CLOS partial integration” on page 218.

The arguments `lisp-class-p`, `lisp-supers` and `lisp-name` control whether the importing also defines a class. `lisp-name` and either `lisp-supers` or `lisp-class-p` must be non-nil to generate a Lisp class.

`lisp-name`, when non-nil, defines the name of the class. Note that by default `lisp-name` is not `nil`, because `generate-java-class-definitions` generates a symbol if `lisp-name` is not supplied.

If `lisp-supers` is supplied and non-nil (and `lisp-name` is non-nil) a class is created, using a plain `defclass` form, and the value `lisp-class-p` is ignored. `lisp-supers` must be a list of symbols naming classes, of which at least one is `standard-java-`
object or a subclass of it. This list defines the superclasses of the class that is defined.

If lisp-class-p is non-nil and lisp-supers is nil (and lisp-name is non-nil) a class is created using ensure-lisp-classes-from-tree. If lisp-class-p is the keyword :complete, the force-p argument is passed is t, otherwise it is nil. See ensure-lisp-classes-from-tree for details.

generate-java-class-definitions returns three arguments: the list of definitions, the lisp-name and the package name.

Notes

generate-java-class-definitions require running Java Virtual Machine, and access to the class definition via the java.lang.reflect package functionality.

See also

write-java-class-definitions-to-stream
write-java-class-definitions-to-file
import-java-class-definitions
“Importing classes” on page 201

get-host-java-virtual-machine

Function

Summary

Return the host Java virtual machine in a dynamic library loaded by Java.

Package

lw-ji

Signature

get-host-java-virtual-machine => jvm

Values

jvm A java-vm-poi or nil.

Description

The function get-host-java-virtual-machine returns the host Java virtual machine when it is called in a dynamic library that was delivered with a call setup-deliver-
dynamic-library-for-java, and the dynamic library was loaded by Java. In all other circumstances it returns nil.

If the initialization of the Java interface is synchronous, which is determined the asynchronous argument of setup-deliver-dynamic-library-for-java and is the default, then during the call to the deliver startup function (the first argument of deliver), get-host-java-virtual-machine still returns nil. It is guaranteed to return the correct value only when the function argument of setup-deliver-dynamic-library-for-java (if any) is called. In the asynchronous case, get-host-java-virtual-machine always returns the correct value.

The result jvm, when it is not nil, is an object of type java-vm-poi.

get-host-java-virtual-machine is useful as a predicate to determine if the library was loaded by Java or non-Java code.

Notes

get-host-java-virtual-machine can find the virtual machine because Java calls JNI_OnLoad with it. If a non-Java code calls JNI_OnLoad with something else, then get-host-java-virtual-machine will return that something else.

When init-java-interface is called without a specified Java virtual machine, it uses get-host-java-virtual-machine to try to find the current one.

See also

setup-deliver-dynamic-library-for-java
init-java-interface
Chapter 15, “Java interface”

get-java-virtual-machine

Function

Summary

If a Java virtual machine has started, return it.

Package

lw-ji
get-java-virtual-machine

Signature  get-java-virtual-machine => java-virtual-machine

Arguments  None.

Values  java-virtual-machine A java-vm-poi.

Description  The function get-java-virtual-machine returns the Java virtual machine if it has started. If LispWorks already knows what the virtual machine is, it just returns it. Otherwise, it tries to use the C function JNI_GetCreatedJavaVMs to try to find it.

See also  init-java-interface

get-jobject

Summary  Get the jobject of the argument.

Package  lw-ji

Signature  get-jobject object => jobject

Signature  ensure-is-jobject object caller => jobject

Arguments  object A Lisp object.

caller A Lisp object.

Values  jobject A jobject.

Description  The functions get-jobject and ensure-is-jobject both get the jobject of the object argument. If object is already a jobject it is simply returned. If it is an instance of standard-java-object and has an associated jobject, this jobject is returned.
Otherwise, \texttt{get\_object} returns \texttt{nil} but \texttt{ensure\_is\_object} signals an error. \texttt{ensure\_is\_object} uses \texttt{caller} in the error message to identify where the error occurred.

Notes

\texttt{get\_object} is the predicate to check whether an object is a Java object.

See also

\texttt{object\_p}

\texttt{object}

“Types and conversion between Lisp and Java” on page 198

“CLOS partial integration” on page 218

\textbf{get\_primitive\_array\_region}

\textbf{set\_primitive\_array\_region}

\textit{Functions}

Summary

Copy between a Java array of primitive type and a buffer specified by a foreign pointer.

Package

\texttt{lw\_ji}

Signature

\texttt{get\_primitive\_array\_region array \&key start end buffer buffer\_size => target\_buffer, foreign\_type}

\texttt{set\_primitive\_array\_region array buffer \&key start end => t, foreign\_type}

Arguments

\textit{array} A Java array of primitive type.

\textit{start, end} Bounding index designators for \textit{array}.

\textit{buffer} An FLI pointer.

\textit{buffer\_size} A non-negative integer.

Values

\textit{target\_buffer} \textit{buffer} or a new buffer.

\textit{foreign\_type} A foreign type.
### Description

The function `get-primitive-array-region` copies from a Java array of primitive type to a buffer specified by a foreign pointer.

The function `set-primitive-array-region` copies from a buffer specified by a foreign pointer to a Java array of primitive type.

*buffer*, if supplied, must be a foreign pointer pointing to a suitable buffer, which means large enough to receive the data in `get-primitive-array-region`, or containing the desired data in `set-primitive-array-region`.

*start* and *end* are bounding index designators for `array`, specifying the region to copy in number of elements.

*buffer-size* is used only when *buffer* is also supplied. *buffer-size* specifies the number of bytes to copy into *buffer*. If copying the required number of elements requires more bytes, `get-primitive-array-region` signals an error. Note that *buffer-size* is specified in bytes, while *start* and *end* are specified in elements.

If *buffer* is not supplied to `get-primitive-array-region` it creates a buffer of the correct size using `fli:allocate-foreign-object`. In this case you will need to free the buffer using `fli:free-foreign-object` when the program has finished with it.

`get-primitive-array-region` copies the required number of elements into the buffer, and returns two values: the target buffer (either the *buffer* argument or the new buffer) and the *foreign-type* corresponding to the Java primitive type (one of `jbyte`, `jshort`, `jint`, `jlong`, `jfloat`, `jdouble`, `jboolean` and `jchar`).

`set-primitive-array-region` copies the required number of elements from *buffer* to `array`, and returns two values: *t* and the foreign type.
Notes
These functions are useful when you need to pass the data to foreign code. If you need the data in Lisp, use \texttt{lisp-array-to-primitive-array} or \texttt{primitive-array-to-lisp-array} instead.

See also \texttt{lisp-array-to-primitive-array} \texttt{primitive-array-to-lisp-array} “Working with Java arrays” on page 210

get-superclass-and-interfaces-tree \textit{Function}

Summary
Returns the superclasses and implemented interfaces of a supplied Java class.

Package \texttt{lw-ji}

Signature
\texttt{get-superclass-and-interfaces-tree java-class => java-class-tree}

Arguments \texttt{java-class} \hspace{1em} A \texttt{jobject}.

Values \texttt{java-class-tree} \hspace{1em} A tree.

Description
The function \texttt{get-superclass-and-interfaces-tree} takes a Java class and returns its superclasses and implemented interfaces. It is used by the importing interface to generate a tree which is then output as argument to \texttt{ensure-lisp-classes-from-tree}. It may be useful on its own, as a quick way of finding the tree for a class.

\texttt{java-class} must be a Java class, that is a \texttt{jobject} corresponding to a class. Typically that would be the result of \texttt{find-java-class}, but it can be the result of your calls to Java methods. Using the Java methods "getInterfaces", "getSuperclass" and "getName" in the Java class "java.lang.Class", \texttt{get-superclass-and-interfaces-tree} constructs a complete
tree of the superclasses and implemented interfaces of the class and its superclasses.

Each node in the tree is a vector of three elements:

0) The full name of the class as a string

1) A node for the superclass (in Java terminology, the one it extends), or nil if there is no superclass (for java.lang.Object and interfaces).

2) A list of nodes corresponding to the interfaces that the class implements.

get-superclass-and-interfaces-tree returns the node for the class itself.

See also ensure-lisp-classes-from-tree
generate-java-class-definitions

import-java-class-definitions

**Macro**

**Summary**
Generates all the definitions for a Java class.

**Package**
1w-ji

**Signature**
import-java-class-definitions java-class-name &key lisp-class-p lisp-name export-p package-name name-constructor lisp-supers => lisp-name-symbol

**Arguments**
java-class-name A string.
lisp-class-p A generalized boolean.
lisp-name A symbol.
export-p A generalized boolean.
package-name A package designator.
name-constructor A function designator.
lisp-supers A list of symbols.
Values  

\[ \text{lisp-name-symbol} \]

A symbol.

Description  

The macro \text{import-java-class-definitions} generates all the definitions for the class \text{java-class-name}, and wraps \text{cl:progn} around them, and returns this from the macroexpansion. Therefore evaluation of an \text{import-java-class-definitions} form defines all the callers for \text{java-class-name}.

\text{java-class-name} name must name a Java Class, and it must be the precise full name.

The generation of the definitions is done by \text{generate-java-class-definitions}, and all the keywords are passed to it. See the entry for \text{generate-java-class-definitions} for the effects of the keywords.

During macroexpansion, \text{import-java-class-definitions} needs to be able to find the class definitions, for which it needs running Java, which means a Java Virtual Machine running and the class being accessible. The evaluation of the definitions does not require Java. Thus if you compile a file containing an \text{import-java-class-definitions} form, the binary file can be loaded without Java, but the compilation needs running Java, and loading the source file also requires running Java.

The returned value \text{lisp-name-symbol} is the Lisp name (which is \text{lisp-name}, or is generated by \text{generate-java-class-definitions} if \text{lisp-name} was not supplied).

See “Importing classes” on page 201 for discussion.

Notes  

1. You can avoid the need for running Java during compilation by writing the definitions using the writers (\text{write-java-class-definitions-to-file} or \text{write-java-class-definitions-to-stream}) once, and incorporate the output into your sources.
2. Even when you use `import-java-class-definitions`, it is probably useful to look at the output of the writers to have a better idea what is actually being generated.

Examples

```lisp
(import-java-class-definitions "java.io.File")
```

See also

`generate-java-class-definitions`
`write-java-class-definitions-to-file`
`write-java-class-definitions-to-stream`
“Importing classes” on page 201

**init-java-interface**

**setup-java-interface-callbacks**

*Functions*

**Summary**

Initializes the Java interface.

**Package**

lw-ji

**Signature**

`init-java-interface &key jvm-library-path java-class-path option-strings jni-env-finder java-virtual-machine class-finder class-loader-finder java-to-lisp-debugger-hook report-error-to-java-host send-message-to-java-host => result`

`setup-java-interface-callbacks &key class-finder java-to-lisp-debugger-hook report-error-to-java-host send-message-to-java-host => t`

**Arguments**

- `jvm-library-path` A string, `t` or `nil`.
- `java-class-path` A string or a list of strings.
- `option-strings` A list.
- `jni-env-finder` A function designator, or `nil`.
- `java-virtual-machine` A foreign pointer of type `java-vm-poi`, or `t`.
- `class-finder` A function designator or `nil`.
- `class-loader-finder` A function designator or `nil`. 
java-to-lisp-debugger-hook

A function designator or \texttt{nil}.

report-error-to-java-host

A function designator or \texttt{nil}.

send-message-to-java-host

A function designator or \texttt{nil}.

Values

\texttt{result} \texttt{t} or the keyword \texttt{:no-java-to-lisp}.

Description

The function \texttt{init-java-interface} needs to be called before using any of the run time part of the Java interface. That includes the interface functions that are documented as requiring Java, and any of the user-defined callers. The definers in general do not need running Java, but the importing interface does.

Note: On Android and in dynamic libraries that were delivered with \texttt{setup-deliver-dynamic-library-for-java} with \texttt{init-java} true (the default), \texttt{init-java-interface} is called automatically by the system initialization, so you do not need to (and must not) call it.

\texttt{init-java-interface} may be used to first initialize the Java Virtual Machine (JVM) or can be called with the JVM already running.

Initializing the virtual machine

If \texttt{init-java-interface} needs to initialize the JVM, it must be called with \texttt{jvm-library-path} either \texttt{t} or the path of a dynamic library, and \texttt{jni-env-finder} must be \texttt{nil}. When \texttt{jvm-library-path} is \texttt{t}, \texttt{init-java-interface} uses a default JVM library path, which is currently "/System/Library/Frameworks/JavaVM.framework/JavaVM" on Mac OS X and ":ljvm" on other Unix variants. On Windows, \texttt{init-java-interface} checks the registry for the location of the JVM library, using the keys that Oracle document in \textit{Java 2 Runtime}.
Environment for Microsoft Windows. The library must implement the JVM, which means exporting the JNI functions, and to be able to find any supporting files that it may need. It loads this library by `fli:register-module`, and then initializes it using `JNI_CreateJavaVM`. The keyword arguments `java-class-path` and `option-strings` can be used to pass options to `JNI_CreateJavaVM`. Except on Mac OS X, passing `jvm-library-path` can work only if the library path contains the JVM library.

`java-class-path` specifies the class path(s) for additional classes on top of the system ones. It is used to specify the `-Djava.class.path` option. If `java-class-path` is a string, it is passed as is, and may contain more than path separated by the appropriate separator (`#:` on Unix, `\;` on Windows), for example `"/myhomedir/myjavaclass;/systemdir/systemjavaclasses/"`. If it is a list, each string should be a path. Each path needs to specify either a directory containing JAR files, or a full path of a JAR file.

If you want to make calls from Java to Lisp, you will need to have the Java class `com.lispworks.LispCalls`. `com.lispworks.LispCalls` is defined in the JAR file `lispcalls.jar` which is part of the LispWorks distribution in the `etc` directory, that is `(lispworks-file "etc/lispcalls.jar")`, so this JAR file will have to be on the path. If you develop for Android and want to import Android classes, you will need the `android.jar` on the path too.

`option-strings` can be used to pass options to `JNI_CreateJavaVM`. Each element of `option-strings` is either a string or a cons of two strings. An element which is a string is passed as the option string (slot `optionString` of the `JavaVMOption` struct). For a cons, the car is passed as the option string, and the cdr as the extra info (slot `extraInfo` in the `JavaVMOption`). Note that that you should not use the option `-Djava.class.path` when using the `java-class-path` argument.
java-class-path and option-strings are ignored when init-java-interface is called after the JVM started.

Calling with JVM already running

If init-java-interface is called with the JVM already running, then jom-library-path must be nil, and either jni-env-finder or java-virtual-machine must be supplied as non-nil, except when called inside a dynamic library delivered with setup-deliver-dynamic-library-for-java, when all arguments can be nil.

If jni-env-finder is non-nil then it must be a function of no arguments that returns a pointer to the JNI environment for the current thread. The result of the finder must be a foreign pointer of type jni-env-poi, corresponding to the C pointer JNIEnv*. The finder function needs to cope with being called on any thread and the result needs to be valid until that thread dies, at which point implementing code must deal with eliminating it. In general, this function needs to know how to find the Java virtual machine, and then use the JNI functions AttachCurrentThread or GetEnv.

If jni-env-finder is nil, then java-virtual-machine is used. If java-virtual-machine is t, LispWorks tried to find the Java virtual machine by first calling get-host-java-virtual-machine, and if this returns nil, by calling JNI_GetCreatedJavaVMs. Otherwise, java-virtual-machine is used as the virtual machine and must be a foreign pointer of type java-vm-poi, corresponding to the C type JavaVM*.

When running in a dynamic library delivered with setup-deliver-dynamic-library-for-java, init-java-interface should be called with jom-library-path, jni-env-finder and java-virtual-machine all nil because the Java virtual machine is obtained using get-host-java-virtual-machine in this case.
Notes

The simple option when the JVM is already running is just passing `java-virtual-machine`. However, the function that the system uses, `JNI_GetCreatedJavaVMS`, is a relic from the time when Java allowed more than one Java VM in each process, which it no longer allows. So in principle some day it may be eliminated (Android already does not define it, but on Android the system calls `init-java-interface` with `jni-env-finder`, so this does not matter). On the other hand it is documented in the latest version (8) without any indication that it is deprecated.

You may have a pointer to the Java VM to pass to `init-java-interface` either because you got it from code that started the Java VM (by `JNI_CreateJavaVM`), or by exporting `JNI_OnLoad` from a dynamic library. However, it is not a good idea to export `JNI_OnLoad` as a foreign callable from LispWorks when it is delivered as a dynamic library, because it will have to wait until LispWorks finished initialization. See “Loading a LispWorks dynamic library into Java” on page 217.

By default, `setup-deliver-dynamic-library-for-java` sets up automatic initialization of the Java interface on start-up. You need (and can call) `init-java-interface` in such a dynamic library only if you passed `nil` for the `init-java` argument to `setup-deliver-dynamic-library-for-java`.

If you call `init-java-interface` with `jvm-library-path`, `jni-env-finder` and `java-virtual-machine` all `nil` and `get-host-java-virtual-machine` returns `nil`, then `init-java-interface` returns `nil` rather than give an error.

Description (cont.)

class-finder specifies a class finder function to be used if the normal search fails. It must be a function taking a string argument, and return a `jobject` representing a class for this string (for example, a caller for the method `java.lang.Class.forName` does the right thing). It is useful when the application knows how to find classes which are not visible from the system class loader. On Android,
finder is passed with a function that calls java.lang.Class.forName with the application Class loader, which will find all classes in the application.

class-loader-finder is used when initializing the LispCalls. If it is non-nil, it must be a function of no arguments that returns a ClassLoader jobject. It is called once during initialization, and the result is stored to be used to find the interfaces when initializing a proxy definition. On Android, it is passed with a function that returns the application class finder. You need to be a Java expert to use this option.

java-to-lisp-debugger-hook, when supplied, must be either a function of one argument or nil. When it is a function, it will be called when the debugger is invoked inside a call from Java to Lisp. The argument is a cl:condition object describing the problem. The function needs to do something to inform the user of the problem but not actually interactively, and return. The caller will then return a default value to Java. By default there is a hook that logs a bug form (by log-bug-form) and prints a message to the console. On Android, it is set to a function that logs the error and then invokes the user Java error reporters (set in Java by com.lispworks.Manager.setErrorReporter and com.lispworks.Manager.setGuiErrorReporter, see the entry for setErrorReporter).

report-error-to-java-host, when supplied, must be a function of two arguments, both of which are strings. When it is passed, if the function report-error-to-java-host is called it uses this function to actually do the report. The first argument is assumed to the error string and the second a filename where there is a bug form, or nil. The function should report to the Java host, whatever that actually means. This keyword is used by the Android interface to set a function that calls into the Android Java code and invokes the same user Java error reporters that are used for the debugger hook above.
The system does not call \texttt{report-error-to-java-host} itself, so the context in which the function may be called is defined by your calls to it. However, it is intended to be used in error handlers, which means it should be able to cope with any context. The default function just prints to \texttt{cl:*terminal-io*}, which may be useful enough when just debugging.

\texttt{send-message-to-java-host}, when supplied, must be a function of two arguments: a string which is the message and a keyword, which tells it how to deal with it. The intent is to modify the "messages output" as described for the \texttt{where-keyword} in \texttt{send-message-to-java-host}. The meaning of "messages output" and the actual behavior is up to the function. On Android it is supplied a function that ends up calling the method \texttt{com.LispWorks.Manager.addMessage}. The default function checks the keyword and then writes the string to \texttt{cl:*terminal-io*}, which is probably good enough for testing purposes.

\texttt{init-java-interface} returns either \texttt{t} for success, or \texttt{:no-java-to-lisp} when it is successful but failed to initialize Java-to-Lisp calls, so you cannot call from Java to Lisp or use Lisp proxies. This failure would normally mean that it failed to find the class \texttt{com.lispworks.LispCalls}.

\texttt{setup-java-interface-callbacks} can be called after \texttt{init-java-interface} was called to change the values of \texttt{class-finder}, \texttt{java-to-lisp-debugger-hook}, \texttt{report-error-to-java-host} or \texttt{send-message-to-java-host}. This is useful in the situations where LispWorks performs the call to \texttt{init-java-interface}, which happens in Android and in a dynamic library delivered with \texttt{setup-deliver-dynamic-library-for-java}.

\textbf{See also} Chapter 15, "Java interface"
**intern-and-export-list**

*Function*

**Summary**
Interns strings in a package and exports the resulting symbols.

**Package**
**lw-ji**

**Signature**
`intern-and-export-list symbol-name-list package-name => nil`

**Arguments**
- `symbol-name-list` A list of strings.
- `package-name` A package designator.

**Values**
`intern-and-export-list` returns nil.

**Description**
The function `intern-and-export-list` finds the package specified by `package-name`, interns all the strings in this package, and exports the resulting symbols.

`intern-and-export-list` is a utility function that is used by the importing interface to export symbols by default (when not using `defpackage`).

**See also**
`generate-java-class-definitions`

---

**jaref**

*Function*

**Summary**
Read and set an element in a Java array.

**Package**
**lw-ji**

**Signature**
`jaref array &rest indices => element`

(setf `jaref`) `new-value array &rest indices => new-value`

**Arguments**
- `array` A Java array, of any type.
- `indices` Non-negative integers.
new-value    A valid element for array.

Values

   element    A Lisp object, a *jobject* or nil.

Description

The function *jaref* reads and sets the value of an element in the Java array *array*.

Each of the indices must be an integer in the right range, which means greater than or equal to 0, and less than than the length of the sub-array ("current array" below) for which they are used. There must be at least one index, and the number of indices must be smaller or equal to the array rank (that is, the number of dimensions) of *array*.

new-value must be a valid value to store in *array*. It has the same restrictions as new-value in (setf *jvref*). See the discussion in *jvref* for details.

The operation of *jaref* and (setf *jaref*) is as follows: For each index except the last, load the element from the "current array", which is the array itself for the first index or the element that was loaded for the previous index. When reaching the last index, *jaref* and (setf *jaref*) get or set the element in the "current array" the same way that *jvref* does. Note that this means that if there are less indices than number of the dimensions of the array, the access will be for a sub-array rather than actual element.

Notes

Because *jaref* needs to load the sub-srray for each index except the last, repeated calls to *jaref* for elements inside the same array are wasteful. It is much more efficient to get the sub-array and access it using *jvref*, or the multiple access functions.

See also

: *jvref*

map-java-object-array
primitive-array-to-lisp-array
lisp-array-to-primitive-array
“Working with Java arrays” on page 210
`java-array-element-type`  

**Function**

**Summary**
Returns the element type of a Java array.

**Package**
lw-ji

**Signature**
`java-array-element-type object => result`

**Arguments**
`object`  
A Java object.

**Values**
`result`  
A keyword, t or nil.

**Description**
The function `java-array-element-type` returns the element type of `object` if `object` is a Java array:

- For primitive types, `result` is a keyword from the table in "Types and conversion between Lisp and Java" on page 198.
- If the array is multi-dimensional, `result` is :array.
- If the array element type is `java.lang.Object`, `result` is :object.
- If the array element type is `java.lang.String`, `result` is :string.
- For other arrays `result` is t.

If `object` is some other type of Java object, `java-array-element-type` returns nil. Otherwise it signals an error.

**Notes**

1. `java-array-element-type` is designed to be fast, so it can be used as a predicate to test whether a Java object is an array, and also to check whether some specific operations are applicable to it.

2. If you want to check whether the array is primitive or not, use `java-primitive-array-element-type` or `java-object-array-element-type` instead.

java-array-error

**Summary**
An abstract class, meaning that it is not signaled. Its readers can be used to access the subclasses.

**Package**
lw-ji

**Superclasses**
java-interface-error

**Subclasses**
java-array-simple-error
java-out-of-bounds-error
java-storing-wrong-type-error

**Readers**
java-array-error-caller
java-array-error-array

**Description**
The condition class java-array-error is an abstract class, meaning that it is not signaled. Its readers can be used to access the subclasses.

java-array-indices-error

**Summary**
Conditions signaled by jaref or (setf jaref) when either too many indices are supplied for the array, or when a sub-array is null.

**Package**
lw-ji

**Superclasses**
java-array-simple-error

**Subclasses**
None.
Readers
java-array-indices-error-rank
java-array-indices-error-indices

Description
Conditions of class java-array-indices-error are signaled when by jaref or (setf jaref) when either too many indices are supplied for the array, or when a sub-array is null.

Notes
You can use the java-array-error readers java-array-error-caller and java-array-error-array on these conditions.

See also
java-array-error
java-out-of-bounds-error

java-array-length

Function

Summary
Returns the length of a Java array.

Package
lw-ji

Signature
java-array-length object => result

Arguments
object
A Java object.

Values
result
A non-negative integer or nil.

Description
The function java-array-length returns the length of object if this is a Java array. For multi-dimensional arrays, java-array-length returns the first dimension. If object is some other type of Java object, java-array-length returns nil. Otherwise it signals an error.

See also
java-array-element-type
“Working with Java arrays” on page 210
java-array-simple-error  

**Condition**

**Summary**  Conditions signaled in miscellaneous array errors.

**Package**  lw-ji

**Superclasses**  java-array-error

**Subclasses**  java-array-indices-error

**Description**  The condition class java-array-simple-error is a subclass of java-array-error, signaled in miscellaneous array errors (those resulting from bad arguments).

**Notes**  You can use the java-array-error readers java-array-error-error-caller and java-array-error-array on these conditions.

**See also**  java-array-error

java-bad-jobject  

**Condition**

**Summary**  An abstract class, meaning that it is not signaled. Its readers can be used to access the subclasses.

**Package**  lw-ji

**Superclasses**  None.

**Subclasses**  java-not-a-java-object-error  
java-instance-without-object-error  
java-not-an-array-error

**Readers**  java-bad-jobject-caller  
java-bad-jobject-object
The condition class `java-bad-object` is an abstract class, meaning that it is never signaled. Its subclasses are signaled and its readers can be used to access those conditions.

See also
- `java-not-a-java-object-error`
- `java-instance-without-jobject-error`
- `java-not-an-array-error`
- `java-definition-error`
- `java-class-error`
- `java-method-error`
- `java-field-error`

### Conditions

**Summary**

Conditions that are signaled when code defined by the Java interface fails to execute.

**Package**

`lw-ji`

**Superclasses**

`java-interface-error`

**Subclasses**

None.

**Readers**

- `java-definition-error-class-name`
- `java-definition-error-name`

For `java-method-error` only:

- `java-method-error-method-name`
- `java-method-error-args-num`

For `java-field-error` only:

- `java-field-error-field-name`
- `java-field-error-static-p`

**Description**

The condition classes `java-class-error`, `java-method-error` and `java-field-error` are signaled when code that is defined by the Java interface (for example, `define-java-`
caller, define-field-accessor) fails to execute because the Java entity that it expects is not found.

They are subclasses of java-definition-error. java-definition-error is never signaled, and you should not signal these conditions.

The errors normally occur because the definition is wrong in some sense, but they can also happen if the Java virtual machine misses some of the classes or has a class definition that differs from what it should be.

java-definition-error-name returns the name of the Lisp function that fails, for example the name in the define-java-caller form.

java-definition-error-class-name returns the class name in the definition.

java-method-error-method-name returns the method name. If it is a constructor (define-java-constructor), java-method-error-method-name returns nil.

java-method-error-args-num returns the number of arguments that were passed to the call.

java-field-error-field-name returns the field name.

java-field-error-static-p queries whether the field was defined as static.

See also define-java-constructor
 define-java-caller
 define-field-accessor

java-exception

Summary The superclass of all conditions that are signaled for Java exceptions.


41 The LW-JI Package

Package lw-ji

Superclasses None

Subclasses java-normal-exception
     java-serious-exception

Readers java-exception-string
       java-exception-exception-name
       java-exception-java-backtrace

Description The class java-exception is the superclass of all conditions
that are signaled for Java exceptions.

The reader java-exception-string returns a string specifying the reason for the exception (result of jobject-string
on the Java exception).

The reader java-exception-exception-name returns a
string with the exception name (name of the exception class,
the result of jobject-class-name on the Java exception).

The reader java-exception-java-backtrace returns a list
of strings specifying the Java backtrace for the exception.
Each string shows one Java frame.

java-field-setting-error Condition

Summary Conditions signaled when setting a field is wrong, either
because the field is final or because the value supplied is
wrong.

Package lw-ji

Superclasses None.

Subclasses None.
Readers

java-field-setting-error-field-name
java-field-setting-error-class-name
java-field-setting-error-class-name-for-setting
java-field-setting-error-new-value

Description

Conditions of class java-field-setting-error are signaled when setting a field is wrong, either because the field is final or because the value supplied is wrong. The setting can happen either by a call to set-java-field or by using (setf name) where name was defined by either define-field-accessor or setup-field-accessor.

The new value returned by the accessor java-field-setting-error-new-value can be the keyword :is-final, which indicates that the error occurs because the field is final. Otherwise it is the new value, which is of an unacceptable type. The class-name of the field can be read using java-field-setting-error-class-name-for-setting (this is what java-field-class-name-for-setting would return for the same field).

See also

set-java-field
define-field-accessor
setup-field-accessor

java-id-exception

Condition

Summary
Conditions signaled if Lisp failed to find the ID for a method or a field.

Package
lw-ji

Superclasses
java-serious-exception

Subclasses
None.
Description  Conditions of class `java-id-exception` are signaled if Lisp failed to find the ID for a method or a field. This is serious, so applications should save and exit.

java-instance-without-jobject-error  Condition

Summary  Conditions signaled when an argument that need to be a Java object is an instance of `standard-java-object` that does not have a `jobject`.

Package  lw-ji

Superclasses  `java-bad-jobject`

Subclasses  None.

Description  The condition class `java-instance-without-jobject-error` is signaled when an argument that needs to be a Java object is an instance of `standard-java-object` that does not have a `jobject`.

Notes  You can use the `java-bad-jobject` readers `java-bad-jobject-caller` and `java-bad-jobject-object` on these conditions.

See also  `java-bad-jobject`
           `java-not-a-java-object-error`
           `java-not-an-array-error`

java-interface-error  Condition

Summary  The superclass of the `*-error` conditions in the Java interface.

Package  lw-ji
Superclasses  cl: error
Subclasses  call-java-method-error
            create-java-object-error
            java-array-error
            java-bad-jobobject
            java-definition-error
            java-field-setting-error
Description  The condition class java-interface-error is the superclass of the *-error conditions in the Java interface.

java-low-level-exception
Condition
Summary  Conditions signaled for failures in low level code.
Package   lw-ji
Superclasses   java-exception
Subclasses  None.
Description  Conditions of class java-low-level-exception are signaled for failures in low level code. This is serious, so applications should save and exit.

java-method-exception
Condition
Summary  Conditions signaled when an exception occurs inside a call to a Java method or constructor.
Package   lw-ji
Superclasses   java-normal-exception
Subclasses

None.

Readers

java-method-exception-name
java-method-exception-class-name
java-method-exception-method-name
java-method-exception-args

Description

The condition class java-method-exception is signaled when an exception occurs inside a call to a Java method or a constructor. Such exceptions are normal behavior for Java, so these exceptions should in general be handled somehow.

The java-exception accessors (java-exception-exception-name, java-exception-string) can be used on a java-method-exception and are useful for simple handling. For more complex handling, you can use catching-java-exceptions around pieces of your code, and then look at the actual Java exception.

The reader java-method-exception-name returns the name of the Java caller (a Lisp symbol) that caused the exception.

The reader java-method-exception-class-name returns the Java class name of the method or constructor.

The reader java-method-exception-method-name returns the method name if the exception is inside a method, or nil if the exception is inside a constructor.

The reader java-method-exception-args returns the arguments that were passed to the caller.

See also

catching-java-exceptions
java-exception

java-normal-exception

Condition

Summary

This condition is signaled for a "normal" exception.
**java-not-a-java-object-error**

*Condition*

**Summary**

Conditions signaled when an argument that needs to be a Java object is not a \texttt{object} or an instance of \texttt{standard-java-object}.

**Package**

lw-ji

**Superclasses**

java-bad-jobject

**Subclasses**

None.

**Description**

The condition class \texttt{java-not-a-java-object-error} is signaled when an argument that needs to be a Java object is not a \texttt{object} or an instance of \texttt{standard-java-object}.

**Notes**

You can use the \texttt{java-bad-jobject} readers \texttt{java-bad-jobject-caller} and \texttt{java-bad-jobject-object} on these conditions.

**See also**

\texttt{java-bad-jobject}
\texttt{java-instance-without-jobject-error}
\texttt{java-not-an-array-error}
java-not-an-array-error

Condition

Summary Conditions signaled when an argument that needs to be an array of some type is not an array of the expected type.

Package lw-ji

Superclasses java-bad-jobject

Subclasses None.

Description The condition class java-not-an-array-error is signaled when an argument that needs to be an array is not an array, or when an argument that needs to be a primitive array is not a primitive array, or when an argument that needs to be an object array is not an object array.

Notes You can use the java-bad-jobject readers java-bad-jobject-caller and java-bad-jobject-object on these conditions.

See also java-bad-jobject
java-not-a-java-object-error
java-instance-without-jobject-error

*java-null*

Constant

Summary A constant representing a Java null pointer.

Package lw-ji

Initial Value A null Java value.

Description The constant *java-null* represents a Java null pointer.

See also “Java non-primitive objects” on page 200
java-object-array-element-type  

Function

Summary

Returns the element type of a Java array of a non-primitive element type.

Package

lw-ji

Signature

java-object-array-element-type object => result

Arguments

object A Java object.

Values

result One of the keywords :array, :object and :string, or nil.

Description

The function java-object-array-element-type returns the element type (as a keyword as listed in “Types and conversion between Lisp and Java” on page 198) if object is an array with non-primitive element type. If object is some other type of Java object, java-object-array-element-type returns nil. Otherwise it signals an error.

Notes

1. You can use java-object-array-element-type to test whether a Java object is an array of non-primitive element type.

   If you want to check whether object is any array (primitive or not), use java-array-element-type instead. Sometimes java-primitive-array-element-type may be more convenient.

See also

java-array-element-type  
java-primitive-array-element-type  
“Working with Java arrays” on page 210

java-objects-eq  

Function

Summary

Tests whether two objects represent the same Java object.
Package  lw-ji

Signature  java-objects-eq  obj1  obj2  =>  result

Arguments  obj1, obj2  Lisp objects.

Values  result  A boolean.

Description  The function java-objects-eq tests whether obj1 and obj2 represent the same Java object.

See also  jobject-p
          jobject
          “Types and conversion between Lisp and Java” on page 198
          “CLOS partial integration” on page 218

java-out-of-bounds-error
java-storing-wrong-type-error

Conditions

Summary  Errors signaled when bad array indices are passed, or on trying to store a bad value into a Java array.

Package  lw-ji

Superclasses  java-array-error

Subclasses  None.

Description  The condition class java-out-of-bounds-error is signaled when a bad index value is passed to jaref or jvref or their setters, or bad start/end values are passed to map-java-object-array and other functions which access arrays.

The condition class java-storing-wrong-type-error is signaled on an attempt to store value of wrong type into a
Java array by `(setf jvref)`, `(setf jaref)` or `map-java-object-array`.

You can use the `java-array-error` readers `java-array-error-caller` and `java-array-error-array` on these conditions.

### java-primitive-array-element-type

**Function**

**Summary**

Returns the element type of a Java array of a primitive element type.

**Package**

 lw-ji

**Signature**

`java-primitive-array-element-type object => result`

**Arguments**

`object` A Java object.

**Values**

`result` A keyword, `t` or `nil`.

**Description**

The function `java-primitive-array-element-type` returns the element type (as a keyword as listed in “Types and conversion between Lisp and Java” on page 198) if `object` is an array with primitive element type. If `object` is some other type of Java object, `java-primitive-array-element-type` returns `nil`. Otherwise it signals an error.

**Notes**

1. `java-primitive-array-element-type` is designed to be fast, so you can use it to test whether a Java object is an array of primitive element type.

2. If you want to check whether `object` is any array (primitive or not), use `java-array-element-type` instead, Sometimes `java-object-array-element-type` may be more convenient.
See also

java-array-element-type
java-object-array-element-type
“Working with Java arrays” on page 210

java-serious-exception

Summary
Conditions signaled when something in the system is not really not as it should be.

Package
lw-ji

Superclasses
java-exception

Subclasses
java-id-exception
java-low-level-exception

Description
The condition class java-serious-exception is signaled for an exception that is serious, which means something in the system is not really not as it should be. Applications that get this should try to save everything and exit.

In general, these exceptions should not happen, and you should not need to worry about these. If you do get any such exception, please report it with as many details as possible to Lisp Support, following the guidelines at www.lisp-works.com/support/bug-report.html.

java-type-to-lisp-array-type
lisp-array-type-to-java-type

Functions

Summary
Return the Lisp array element type matching a supplied foreign type, or the foreign type matching a Lisp array element type.

Package
lw-ji
Signature: Java

`java-type-to-lisp-array-type jtype => l-result`

Signature: Lisp

`lisp-array-type-to-java-type lisp-type`

Arguments:
- `jtype`: A foreign type.
- `lisp-type`: A Lisp type specifier.

Values:
- `l-result`: A Lisp array element type, or `nil`.

Description:
The function `java-type-to-lisp-array-type` returns the matching Lisp array element type for the foreign type `jtype`, which needs to be one of the foreign types corresponding to a Java primitive type, or `nil` if the argument is not such a foreign type.

The function `lisp-array-type-to-java-type` returns the matching foreign type, corresponding to a Java primitive type, for the Lisp array element type `lisp-type`, or `nil` if there is no match.

Both functions use the table below for doing the match:

<table>
<thead>
<tr>
<th>Foreign type</th>
<th>Lisp type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jbyte</code></td>
<td>(signed-byte 8)</td>
</tr>
<tr>
<td><code>jshort</code></td>
<td>(signed-byte 16)</td>
</tr>
<tr>
<td><code>jint</code></td>
<td>(signed-byte 32)</td>
</tr>
<tr>
<td><code>jlong</code></td>
<td>(signed-byte 64)</td>
</tr>
<tr>
<td><code>jdouble</code></td>
<td>double-float</td>
</tr>
<tr>
<td><code>jfloat</code></td>
<td>single-float</td>
</tr>
<tr>
<td><code>jchar</code></td>
<td>(unsigned-byte 16)</td>
</tr>
<tr>
<td><code>jboolean</code></td>
<td>(unsigned-byte 8)</td>
</tr>
</tbody>
</table>
The LW-JI Package

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java-vm-poi</td>
<td>The FLI type Java virtual machine pointer (JavaVM* in C). You need it only when you want to pass the virtual machine to init-java-interface by the java-virtual-machine argument.</td>
</tr>
</tbody>
</table>

See also: init-java-interface

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jboolean</td>
<td>FLI types corresponding to the Java primitive types.</td>
</tr>
<tr>
<td>jbyte</td>
<td>FLI types corresponding to the Java primitive types.</td>
</tr>
<tr>
<td>jchar</td>
<td>FLI types corresponding to the Java primitive types.</td>
</tr>
<tr>
<td>jdouble</td>
<td>FLI types corresponding to the Java primitive types.</td>
</tr>
<tr>
<td>jfloat</td>
<td>FLI types corresponding to the Java primitive types.</td>
</tr>
<tr>
<td>jint</td>
<td>FLI types corresponding to the Java primitive types.</td>
</tr>
<tr>
<td>jlong</td>
<td>FLI types corresponding to the Java primitive types.</td>
</tr>
<tr>
<td>jshort</td>
<td>FLI types corresponding to the Java primitive types.</td>
</tr>
</tbody>
</table>

See “Types and conversion between Lisp and Java” on page 198 for discussion.
**jni-env-poi**

*FLI type descriptor*

Summary  The JNI environment pointer type.

Package  lw-ji

Syntax  jni-env-poi

Description  The FLI type jni-env-poi is used for the JNI environment pointer (JNIEnv* in C).

When jni-env-finder is passed to init-java-interface, it needs to be a function that returns a jni-env-poi.

See also  init-java-interface

---

**jobject**

*FLI type descriptor*

Summary  The type of objects representing all non-primitive Java objects.

Package  lw-ji

Syntax  jobject

Description  The FLI type jobject is the type of objects representing all non-primitive Java objects (including arrays).

jobjects that represent the same Java object are not necessarily equal in any Lisp sense, and their addresses are not necessarily equal either. In fact, normally they will be different if they come from a different Java call. To check whether two jobjects represent the same Java object, use java-objects-eq (which takes CLOS Java instances too).

Notes  The print-function of jobject tries to print its Java class name, but what it prints may be a parent class of the actual
class of the *object*. The function *jobject-class-name* returns the name of the actual class of the *object*, and also caches it in the *object*.

See also *jobject-p*  
  *java-objects-eq*  
  *jobject-string*  
  *jobject-class-name*  
  *jobject-of-class-p*  
  *jobject-to-lisp*  
  *jobject-pretty-class-name*  
  “Types and conversion between Lisp and Java” on page 198

---

**jobject-class-name**  

*Function*

**Summary**  
Returns the name of the class to which a *object* belongs.

**Package**  
*lw-ji*

**Signature**  
*jobject-class-name object => class-name*

**Arguments**  
*object*  
A *object*.

**Values**  
*class-name*  
A string.

**Description**  
The function *jobject-class-name* returns a string which is the name of the class to which the Java object *object* belongs. The name is then cached in the *object*.

The class for arrays is the internal class name, which is different from the way it is declared in Java. For other objects, the name is the full name of the class.

To obtain the class name as declared in Java, use *jobject-pretty-class-name*. 
See also  
\texttt{jobject}  
\texttt{jobject-of-class-p}  
\texttt{jobject-pretty-class-name}  
“Types and conversion between Lisp and Java” on page 198

\textbf{\texttt{jobject-ensure-global}}  
\textit{Function}  

\textbf{Summary}  
Returns a global \texttt{jobject} pointing to the same Java object as the argument.

\textbf{Package}  
\texttt{lw-ji}  

\textbf{Signature}  
\texttt{jobject-ensure-global} \texttt{\texttt{jobject} => global-jobject}  

\textbf{Arguments}  
\texttt{jobject}  
\texttt{A \texttt{jobject}.}  

\textbf{Values}  
\texttt{global-jobject}  
\texttt{A \texttt{jobject}.}  

\textbf{Description}  
The function \texttt{jobject-ensure-global} returns a \texttt{jobject} pointing to the same Java object as the argument \texttt{jobject}, but which is guaranteed to be global.

In most cases, \texttt{jobjects} are global anyway. However, when using \texttt{map-java-object-array}, by default, the \texttt{jobjects} are local and cannot be used outside the scope of the function that was passed to \texttt{map-java-object-array}. Similarly, \texttt{jobjects} can be made local inside functions that are invoked by proxies, using the \texttt{:jobject-scope} option (see \texttt{define-lisp-proxy}). In these situations, if you want to access the Java object outside the scope of the function that was invoked by \texttt{map-java-object-array} or by the proxy, you need to use \texttt{jobject-ensure-global inside} the scope of the function, and then you can use the result outside the scope of the function.

If the argument \texttt{jobject} is not a \texttt{jobject} an error is signaled.
If the argument *object* is already a global reference, \texttt{object-ensure-global} simply returns it.

Notes

1. \texttt{object-ensure-global} cannot access local references outside the right scope (like any other function).

2. \texttt{object-ensure-global} does not accept an instance of \texttt{standard-java-object}.

See also

- \texttt{object-p}
- \texttt{map-java-object-array}
- \texttt{define-lisp-proxy}

**jobobject-of-class-p**

Function

Summary

The predicate for whether a Java object is an instance of a given Java class.

Package

\texttt{lw-ji}

Signature

\texttt{jobobject-of-class-p object class-spec \rightarrow result}

Arguments

- *object*: A Java object.
- *class-spec*: A class specifier that \texttt{find-java-class} accepts or a Java class.

Values

- *result*: A generalized boolean.

Description

The function \texttt{jobobject-of-class-p} takes a Java object and a Java class specification, and returns true if the object is an instance of this class or any of its subclasses.

*class-spec* must be either a class specifier that \texttt{find-java-class} accepts, or a Java class, that is a \texttt{Object} of class \texttt{java.lang.Class}. The Java class may be an interface, in which case the result verifies whether the object implements the interface.
See also  
**jobject**  
**jobject-class-name**  
“Types and conversion between Lisp and Java” on page 198

---

### jobject-p

**Function**

**Summary**
The predicate for objects of type **jobject**.

**Package**
**lw-ji**

**Signature**
`jobject-p object => result`

**Arguments**
- `object` | A Lisp object.

**Values**
- `result` | A boolean.

**Description**
The function **jobject-p** is the predicate for objects of type **jobject**.

See also
- `jobject`
- `lisp-java-instance-p`
- `get-jobject`
- `ensure-is-jobject`
- “Types and conversion between Lisp and Java” on page 198

---

### jobject-pretty-class-name

**Function**

**Summary**
Returns a string which is the name of the class to which a given **jobject** belongs.

**Package**
**lw-ji**

**Signature**
`jobject-pretty-class-name object => name`

**Arguments**
- `object` | A **jobject**.

---
The function `jobject-pretty-class-name` returns a string which is the name of the class to which the `jobject` object belongs. The name is then cached in the `jobject`.

The class for arrays is "prettified", which means converting it to the way it is declared in Java. For other objects, `name` is the same as the result of `jobject-class-name`.

See also

- `jobject`
- `jobject-class-name`
- “Types and conversion between Lisp and Java” on page 198

---

**jobject-string**

*Function*

**Summary**
Calls the Java method `Object.toString` on a Java object.

**Package**
`lw-ji`

**Signature**
```
jobject-string  jobject  =>  string
```

**Arguments**
- `jobject` A `jobject`.

**Values**
- `string` A string.

**Description**
The function `jobject-string` returns a string which is the result of calling the Java method `Object.toString` on the Java object `jobject` on it.

See also

- `jobject`
- “Types and conversion between Lisp and Java” on page 198
**jobject-to-lisp**

**Summary**
Converts a **object** to a Lisp object where possible.

**Package**
lw-ji

**Signature**

```
jobject-to-lisp object &optional nil-when-fail => lisp-object
```

**Arguments**

- **object**
  A **object** or **nil**.

- **nil-when-fail**
  A generalized boolean.

**Values**

- **lisp-object**
  A Lisp object.

**Description**
The function **jobject-to-lisp** converts a **object** to a Lisp object where possible.

The argument **object** must be a **object** or **nil**, otherwise an error is signaled. If **object** is **nil**, **jobject-to-lisp** returns **nil**. If **object** is a **object** of type **java.lang.String** or any of the primitive types, **jobject-to-lisp** returns the matching Lisp object. See “Types and conversion between Lisp and Java” on page 198 for a full description.

If the conversion cannot be done, the return value depends on the value of **nil-when-fail**. When **nil-when-fail** is true **jobject-to-lisp** returns **nil** for failure. When **nil-when-fail** is false, **jobject-to-lisp** returns the **object** itself. The default value of **nil-when-fail** is true.

**Notes**
You need to pass **nil-when-fail as nil** for the cases when you want to be able to distinguish between return value **nil** for the Java boolean **false** and failure to convert. When you do that, the caller code needs to compare the result to the argument, instead of checking for non-nil, like this:

```
(let ((my-res (jobject-to-lisp my-obj nil)))
  (if (eq my-obj my-res)
      (fail-branch)
      (success-branch)))
```
**jvref**

**Function**

**Summary**
Read and set an element in a Java array.

**Package**
lw-ji

**Signature**

\[
\text{jvref array index => element}
\]

\[
(\text{setf jvref}) \text{ new-value array index => new-value}
\]

**Arguments**

array
A Java array.

index
A non-negative integer.

new-value
A valid value for array.

**Values**

element
A Lisp object, a `object` or `nil`.

**Description**
The function **jvref** reads and sets the value of an element in the Java array `array`.

*index* must be in the right range:

\[
0 \leq index < (\text{java-array-length array})
\]

*new-value* must be a valid value to store in `array` (discussed below)

**jvref** returns the corresponding element from `array`. If the element is of a primitive type, or is of type `java.lang.String`, it is converted to the Lisp object, otherwise it is returned as a `object` or `nil` if it is `null`. See “Types and conversion between Lisp and Java” on page 198.

*(setf jvref)* sets the element to *new-value*. *new-value* must be a valid element for `array`. For a primitive array, *new-value* must be a Lisp object of the correct type:

- byte, short, int, long
  - Integers with less than 8, 16, 32 and 64 bits respectively.

- float, double
Any float.

```lisp
boolean
nil or t.
```

cchar

Integers in the inclusive range \([0, \#ffffff]\).

For a non-primitive array, `new-value` must be convertible to a `jobject` of the correct class. If the element type of `array` is `java.lang.Object` (``java-array-element-type`` returns :object), then any Lisp value that can converted to a Java primitive type is acceptable (see “Types and conversion between Lisp and Java” on page 198), as well as strings and any Java object. If the element type of `array` is `java.lang.String` (``java-array-element-type`` returns :string), then strings or Java objects of class `java.lang.String` are acceptable. In all other cases, only Java objects are acceptable, and need to be of the correct type.

Notes

For accessing multiple elements in the same array, the multiple access functions (``map-java-object-array``, `primitive-array-to-lisp-array`, `lisp-array-to-primitive-array`) can be much faster.

``jvref`` and (``setf jvref``) access the top level of the array. If `array` is multi-dimensional, `jvref` and (``setf jvref``) will return and set the sub-array. See `jaref` for accessing elements in a multi-dimensional array.

``jvref`` and (``setf jvref``) are slightly faster than `jaref` and (``setf jaref``) with one index, and give a proper error when called with the wrong number of arguments.

See also

``map-java-object-array``
``lisp-array-to-primitive-array``
``primitive-array-to-lisp-array``
``jaref``

“Working with Java arrays” on page 210
**lisp-java-instance-p**  
*Function*

**Summary**  
The predicate for objects of type `standard-java-object`.

**Package**  
lw-ji

**Signature**  
lisp-java-instance-p object => result

**Arguments**  
object A Lisp object.

**Values**  
result A boolean.

**Description**  
The function `lisp-java-instance-p` is the predicate determining whether an object is an instance of `standard-java-object`.

**See also**  
object-p  
jobobject  
get-jobobject  
ensure-is-jobobject  
“Types and conversion between Lisp and Java” on page 198  
“CLOS partial integration” on page 218

**lisp-to-object**  
*Function*

**Summary**  
Converts a Lisp object to an appropriate jobobject.

**Package**  
lw-ji

**Signature**  
lisp-to-object lisp-object &optional errorp => result

**Arguments**  
lisp-object A Lisp object.

**Values**  
result A jobobject or nil.
The function lisp-to-jobject tries to convert the argument lisp-object to a jobject. It succeeds if lisp-object is of a type that matches any Java primitive type or is a string. In general that means integers up to 64 bits, floats, t, nil and strings.

See “Types and conversion between Lisp and Java” on page 198 for a full description.

If it fails, lisp-to-jobject calls cl: error, unless the argument errorp is nil, in which case it returns nil.

See also  jobject-to-lisp
“Types and conversion between Lisp and Java” on page 198

make-java-array

Function

Create a Java array object.

Signature
make-java-array type first-dim &rest dims => array

Arguments
type A string, one of the keywords :byte, :short, :int, :long, :float, :double, :char, :boolean, :object and :string, an FLI type specifier, or t.

first-dim A non-negative integer.

dims Non-negative integers.

Values
array A new array.

Description
The function make-java-array creates a Java array object array.

type specifies the type of elements in array. To make an array of any Java class, type needs to be a string with the full name of the class. To make an array of primitive type, type should
be the corresponding keyword (:byte, :short, :int, :long, 
:float, :double, :char or :boolean). type can also be 
:object or t meaning java.lang.Object, and :string 
meaning java.lang.String, and the FLI types matching the 
primitive types.

The dimension(s) of the array are specified by first-dim and 
the dims list, which must all be non-negative integer(s).

make-java-array returns the new array.

**make-java-instance**

**Summary**  
Create a CLOS instance and its jobject.

**Package**  
lw-ji

**Signature**  
make-java-instance symbol-or-class &rest args => instance

**Arguments**  
symbol-or-class  A class designator.

args  Lisp objects.

**Values**  
instance  A CLOS object.

**Description**  
The function **make-java-instance** creates a CLOS instance 
and its jobject.

The class symbol-or-class must be a subclass of standard-
java-object, and must have been associated with a Java 
constructor by passing the class name to define-java-con-
structor or setup-java-constructor as the class-symbol 
argument (the importing interface, when defining a class, 
does it automatically).

make-java-instance makes the CLOS instance by calling 
make-instance on symbol-or-class, then passing the instance 
and args to create-instance-jobject-list to create the 
jobject, and then returns the instance.
The result is a CLOS instance of symbol-or-class, which can be passed to Java interface functions and Java methods.

See also

- create-instance-jobject-list
- define-java-constructor
- setup-java-constructor
- "CLOS partial integration" on page 218

### make-lisp-proxy

#### make-lisp-proxy-with-overrides

**Summary**

Make a Lisp proxy.

**Package**

lw-ji

**Signature**

```make-lisp-proxy
name &key user-data print-name overrides
overrides-plist => proxy
```

**Signature**

```make-lisp-proxy-with-overrides
name &rest args &key user-data print-name &allow-other-keys => proxy
```

**Arguments**

- **name**
  - A symbol.
- **user-data**
  - A Lisp object.
- **print-name**
  - A string or a symbol.
- **overrides**
  - An association list.
- **overrides-plist**
  - A plist.
- **args**
  - A plist.

**Values**

- **proxy**
  - A jobject.

**Description**

The functions `make-lisp-proxy` and `make-lisp-proxy-with-overrides` make a Lisp proxy, which is a Java proxy where method invocation ends up calling Lisp code. The result is a jobject proxy which represents the proxy, which
can then be used in Java where an object that implements any of the interfaces that the proxy implements is required.

**Note:** The *object* is "local", which means that if it is generated in the scope of a call from Java it must be used (passed to Java method, return to the call from Java or pass it to *object-ensure-global*) in the scope of the call from Java. You cannot store it in Lisp and use it later (but you can do that with the result of *object-ensure-global*). If the *object* is generated not in the scope of a Java call, it must be used on the same thread that it was made.

*name* must be associated with a proxy definition, either by *define-lisp-proxy* or *setup-lisp-proxy*. The proxy definition determines which interfaces the proxy implements, and what happens when a method is invoked on the proxy. The processing of invocation of a method on the proxy is described in the entry for *define-lisp-proxy*.

*user-data* is an arbitrary object. It is passed to the Lisp function if the proxy definition specifies that it should be passed (keyword *with-user-data* or *default-function-with-user-data* for the default function).

*print-name* specifies the name of the proxy, after it is converted to a string by *cl:princ-to-string*. If the proxy definition has a *print-name* too, the full print name of the proxy is formed by concatenating the definition's *print-name* and the proxy's *print-name* separated by " - ", otherwise the full print name is the proxy's *print-name*. The full print name is used when printing the proxy, and is also returned when the Java method *toString* is applied to it. If *print-name* is *nil*, a counter is used.

*overrides*, if supplied, must be an association list specifying overriding (see “Overriding” below), that is a list of conses where the *cl:car* is the symbol to override and the *cl:cdr* is the target. When *overrides* is non-nil *overrides-plist* is ignored.
If `overrides-plist` is supplied it must be a plist specifying overriding, that is a list of even length where each even element is a symbol to override and the following odd element is the target.

The `args` argument of `make-lisp-proxy-with-overrides` is used as a plist specifying overrides, after removing any occurrences of `:print-name` and `:user-data` from it.

**Overrides**

Overrides allow `make-lisp-proxy` and `make-lisp-proxy-with-overrides` to override symbols in the proxy definition, which means that instead of calling the symbol in the proxy definition the target in the overrides is called. See the entry for `define-lisp-proxy` for details of the processing.

`make-lisp-proxy-with-overrides` is intended to make it simpler to use overrides. It is equivalent to calling `make-lisp-proxy` with `overrides-plist`, and actually calls `make-lisp-proxy` (so may get errors that look like they came from `make-lisp-proxy`).

`make-lisp-proxy` and `make-lisp-proxy-with-overrides` signal error if name is not associated with a proxy definition, and if any overrides are not of the right form or any of the functions to call is not a function designator. They may also signal an error if the proxy definition was not initialized and they failed to initialize it.

See also `define-lisp-proxy`  
`setup-lisp-proxy`  
“Using proxies” on page 207

**map-java-object-array**

**Function**

**Summary**

Apply a function to the elements in an array.
The function `map-java-object-array` applies the function `function` to the elements in the Java array `array`.

The default behavior is simply to apply `function` to each element. The keyword arguments can be used to change this behavior, including modifying elements.

`function` should take one or two arguments, depending on `pass-args`. The default value of `pass-args` is `:element`, which means that `function` takes one argument, the element in the array. `pass-args` can also be `:element-index`, and then `function` should take two arguments, the element and the index. `pass-args` can also be `:index` in which case `function` just takes the index. The latter case is useful when `map-java-object-array` is used to modify the element in the array. When `function` is `nil`, the "result" of the function call is the element itself. That is useful for simple collection (that is, supplying a true value of `collect`).
Note: When the element that is passed to function is a jobj-ect, it is by default a "local" object, which means it must not be used outside the dynamic scope of the function call. collect and convert can change this.

When write-back is nil the result of the call to function is ignored. When write-back is non-nil, the result of function is the new value to write back. The default value of write-back is nil.

When reverse is non-nil map-java-object-array starts from the highest index and maps down, otherwise it maps up. The default value of reverse is nil.

start and end specify the range in array to map: start defaults to 0 and is inclusive, and end defaults to the length of array and is exclusive. If either of these is not an integer or is out of bounds, or end is smaller than start, then an error of type java-out-of-bounds-error is signaled.

pass-args controls the arguments to function as described above.

collect, if non-nil, specifies that the results of applying function should be collected and returned from map-java-object-array. If collect is t, map-java-object-array returns a list of the results. collect can also be :vector or cl:vector, in which case result is a Lisp vector. When convert is either nil or t, collect overrides it and forces conversion of primitive types and strings to Lisp objects, and makes jobjects non-local, so they can be used outside the scope of the function calls and map-java-object-array. The default value of collect is nil.

convert controls conversion to Lisp objects. When convert is t, primitive types and strings are converted to Lisp objects before they are passed to function. When convert is nil, all elements are passed as jobjects. Note that when collect is non-nil and convert is nil or t, collect overrides convert as described above. The default value of convert is t.
When `convert` is one of `:force-nil`, `:force-local` or `:force-global` it overrides `collect`. `:force-nil` causes the object to pass as a `jobject` (the same as when `collect` is `nil` and `convert` is `nil`). `:force-local` causes primitive types to pass as Lisp objects, and other types as local `jobjects` (the same as when `collect` is `nil` and `convert` is `t`). `:force-global` causes primitive types to be passed as Lisp objects and other types as global objects.

**Note:** local `jobjects`, which you get when `convert` is either `:force-nil` or `:force-local`, or when `collect` is `nil` and `convert` is not `:force-global`, must not be used outside the scope of the function `function` that is passed to `map-java-object-array`. Using local objects out of scope can cause the system to crash (rather than signal an error). Note that you must not even use a local `jobject` from one call to `function` in another call to `function` within the same call to `map-java-object-array`.

Converting to global objects adds a substantial overhead to the system, though for small arrays this is not very bad. If you want to map over a large array, and dynamically decide to use only some of the `jobjects` out of scope, you can convert local `jobjects` to global using `jobject-ensure-global`.

When `write-back` is true, the result of the application of `function` is written back to `array`. The default value of `write-back` is `nil`.

If `array` is not a non-primitive Java array, or `pass-args` or `collect` is not one of the acceptable values, or `write-back` is non-nil and `function` returns an object of wrong type, `map-java-object-array` signals an error of type `java-array-error`. 
Notes


2. The function `java-object-array-element-type` can be used to test whether a Java object is a non-primitive array.

3. When accessing more than one element, `map-java-object-array` may be much faster than accessing the elements using `jvref` or `jaref`.

4. `map-java-object-array` traverses one level. If a multi-dimensional array is supplied, the elements that it passes to function are sub-arrays (which may be `null` too).

See also

- `jvref`
- `jaref`
- `primitive-array-to-lisp-array`
- `lisp-array-to-primitive-array`
- `get-primitive-array-region`
- `set-primitive-array-region`
- `java-object-array-element-type`
- `jobject-ensure-global`
- “Working with Java arrays” on page 210

**Functions**

**primitive-array-to-lisp-array**

**lisp-array-to-primitive-array**

**Summary**

Copy elements between a Java primitive array and a Lisp array of matching type.

**Package**

`lw-ji`

**Signature**

`primitive-array-to-lisp-array p-array &key start end target-start target-end lisp-array => l-result`
The LW-JI Package

41

lisp-array-to-primitive-array l-array &key start end target-start target-end primitive-array => p-result

Arguments

p-array A Java array of primitive type.
start, end Bounding index designators.
target-start, target-end Bounding index designators.
lisp-array A Lisp array of an acceptable type, or nil.
l-array A Lisp array of an acceptable type.
primitive-array A Java array of primitive type, or nil.

Values

l-result lisp-array or a new Lisp array.
p-result primitive-array or a new primitive array.

Description

The function primitive-array-to-lisp-array takes a Java array p-array of primitive type and copies elements from it to a Lisp array of matching type. The target lisp-array is created by default, but can also be supplied as an argument.

The function lisp-array-to-primitive-array takes a Lisp array l-array of an acceptable Lisp type and copies elements from it to a Java array of matching type. The target primitive-array is created by default, but can also be supplied as an argument.

start and end are bounding index designators for the source p-array or l-array, specifying the range to copy.

target-start and target-end are used only if the target is supplied (by lisp-array or primitive-array). They specify the start and end for copying in the target. The actual number of elements copied is the minimum of the lengths specified for the source and for the target.

If the target (lisp-array or primitive-array) is not supplied, these functions create an array of the correct type and the copy length, and copy into it.

1300
The Lisp array that is passed to `lisp-array-to-primitive-array` must be of one of the types listed below, and when the target array is supplied, its type must match the type of the source array according to the table below, except that `cl:base-char array (cl:simple-base-string)` is acceptable when the Java side is `byte`.

Table 41.2 Correspondence between Java primitive and Lisp array element types

<table>
<thead>
<tr>
<th>Java primitive type</th>
<th>Keyword (result of <code>java-array-element-type</code>)</th>
<th>Lisp type (result of <code>cl:array-element-type</code>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>:byte</td>
<td>(signed-byte 8)</td>
</tr>
<tr>
<td>short</td>
<td>:short</td>
<td>(signed-byte 16)</td>
</tr>
<tr>
<td>int</td>
<td>:int</td>
<td>(signed-byte 32)</td>
</tr>
<tr>
<td>long</td>
<td>:long</td>
<td>(signed-byte 64)</td>
</tr>
<tr>
<td>double</td>
<td>:double</td>
<td>double-float</td>
</tr>
<tr>
<td>float</td>
<td>:float</td>
<td>single-float</td>
</tr>
<tr>
<td>char</td>
<td>:char</td>
<td>(unsigned-byte 16)</td>
</tr>
<tr>
<td>boolean</td>
<td>:boolean</td>
<td>(unsigned-byte 8)</td>
</tr>
</tbody>
</table>

For `boolean`, 1 is `true` and 0 is `false`.

Notes

For a large number of elements, these functions are much faster than `jvref`. If the primitive data is needed for passing to or from foreign functions, use `get-primitive-array-region` and `set-primitive-array-region` instead. These functions work only on arrays with one dimension with
primitive element type. For non-primitive arrays of one dimension you can use `map-java-object-array`.

See also

- `get-primitive-array-region`
- `set-primitive-array-region`
- `map-java-object-array`
- `jvref`
- `jaref`
- “Working with Java arrays” on page 210

**read-java-field**

**checked-read-java-field**

**set-java-field**

**check-java-field**

**java-field-class-name-for-setting**

**Functions**

**Summary**

Access a field, either static or in a Java object.

**Package**

lw-ji

**Signature**

`read-java-field full-field-name &optional object => field-value`

`checked-read-java-field full-field-name &optional object => field-value-or-nil, nil-or-condition`

`set-java-field full-field-name value &optional object => value`

`check-java-field full-field-name static-p => result`

`java-field-class-name-for-setting full-field-name static-p => class-name or nil`

**Arguments**

- `full-field-name` A string.
- `object` A Java object or nil.
- `value` A type.
static-p A boolean.

field-value The value of the field.

field-value-or-nil The value of the field or nil.

nil-or-condition nil or a cl: error.

result A boolean.

Description

The functions read-java-field, checked-read-java-field and set-java-field access the value of a field in a Java Object or a static field.

The functions check-java-field and java-field-class-name-for-setting are used to check whether it is possible to access the value of the field.

full-field-name needs to be a full field name including the package and class, for example "java.io.File.separator".

If object is supplied and is non-nil, it must be a Java object from which to read/to which set the value. The field must be non-static in this case. If object is nil, the field must be static.

read-java-field returns the value of the field. If it fails to get it, it signals an error. If the class is not found, this is a java-class-error, if the field is not found it is a java-field-error.

checked-read-java-field returns the value like read-java-field and another value which is nil when the read is successful. If the class or the field is not found, checked-read-java-field returns nil and a condition specifying the error (java-field-error or java-class-error). Note that it may still signal other errors, for example if full-field-name does not look like a proper field name.

set-java-field sets the field to the value. value must be of an acceptable type (see “Types and conversion between Lisp
and Java” on page 198) and the field must not be final, otherwise it signals java-field-setting-error.

check-java-field checks whether the field exists and matches the value of static-p, and returns a boolean result accordingly.

java-field-class-name-for-setting checks whether the field exists and matches the value of static-p and whether it is not final, and if it is returns the class name of the field. It returns nil otherwise. java-field-class-name-for-setting is useful for checking whether set-java-field can be used on a a field, and whether a value is suitable to be stored in his field, by using jobject-of-class-p.

Notes It is also possible to access fields using accessors defined by define-field-accessor and setup-field-accessor.

See also java-field-error
java-class-error
java-field-setting-error
define-field-accessor
setup-field-accessor
Chapter 15, “Java interface”

record-java-class-lisp-symbol Function

Summary Records a correspondence between the name of a Java class and a Lisp symbol.

Package lw-ji

Signature record-java-class-lisp-symbol java-class-name lisp-name => lisp-name

Arguments java-class-name A string.
lisp-name A symbol.
Values

\[ lisp-name \quad \text{A symbol.} \]

Description

\[ java-class-name \] must be the precise full name of a Java class. \[ lisp-name \] must be a Lisp symbol corresponding to the Java class. The function \texttt{record-java-class-lisp-symbol} records this correspondence.

At the time of writing this correspondence is used only to find CLOS class names from Java class name by \texttt{create-instance-from-jobject} and \texttt{ensure-lisp-classes-from-tree}.

\texttt{record-java-class-lisp-symbol} is used by the importing interface when \[ lisp-name \] is non-nil. You can use it yourself, but \[ lisp-name \] must name an appropriate class (subclass of \texttt{standard-java-object}).

See also

\texttt{write-java-class-definitions-to-stream}  
\texttt{write-java-class-definitions-to-file}  
\texttt{import-java-class-definitions}  
\texttt{create-instance-from-jobject}  
\texttt{ensure-lisp-classes-from-tree}

\textbf{report-error-to-java-host}  

\textit{Function}

Summary

Tries to report an error to the Java host.

Package

\texttt{lw-ji}

Signature

\texttt{report-error-to-java-host error-string log-file-string => result}

Arguments

\[ error-string \quad \text{A string.} \]

\[ log-file-string \quad \text{A string or nil.} \]

Values

\[ result \quad \text{A boolean.} \]
The function `report-error-to-java-host` tries to report an error to the Java host.

It is assumed that `error-string` specifies the error and `log-file-string` specifies a file where a log of the error is written.

`report-error-to-java-host` funcalls the function that was passed as the `report-error-to-java-host` argument to `init-java-interface`, or the default function, with these two arguments.

On Android the keyword argument to `init-java-interface` is passed with a function that invokes the user Java error reporters (set in Java by `com.lispworks.Manager.setErrorReporter` and `com.lispworks.Manager.setGuiErrorReporter`).

The default function just prints to `cl:*terminal-io*`, which may be useful enough for debugging.

`result` is `t` if there is a function, and `nil` otherwise.

See also `init-java-interface`
information. \textit{for-shaking-p} needs to be true when it is called before shaking.

The system automatically calls \texttt{reset-java-interface-for-new-jvm} when delivering, so you do not need to call it then. Applications should never use it. It may be useful during debugging if the JVM is manipulated in some way.

The default value of \textit{for-shaking-p} is \texttt{nil}.

\textbf{send-message-to-java-host}  

\textit{Function}

\textbf{Summary}  
Sends a message to the Java host.

\textbf{Package}  
lw-ji

\textbf{Signature}  
\texttt{send-message-to-java-host message-string where-keyword => result}

\textbf{Arguments}  
message-string  
A string.

where-keyword  
One of the keywords \texttt{append}, \texttt{add}, \texttt{prepend}, \texttt{add-no-scroll}, \texttt{append-no-scroll} and \texttt{reset}.

\textbf{Values}  
result  
A boolean.

\textbf{Description}  
The function \texttt{send-message-to-java-host} sends a message to the Java host. It funcalls the function that was passed as the \texttt{send-message-to-java-host} argument to \texttt{init-java-interface}, or the default function, with \texttt{message-string} and \texttt{where-keyword}.

On Android \texttt{init-java-interface} is given a function that ends up calling the method \texttt{com.lispworks.Manager.addMessage}.  

The default function checks the keyword and then writes the string to `cl:*terminal-io*`, which is probably good enough for testing purposes.

`result` is `t` if there is a function, and `nil` otherwise.

**Notes**
The intended meanings of `where-keyword` are:

- **:reset** Erase any existing text and replace it by the message.
- **:prepend** Insert the message and a newline before any existing text.
- **:add-no-scroll** Add the message after all existing text.
- **:add** Like `:add-no-scroll`, and scroll to the beginning of the new message.
- **:append-no-scroll** Like `:add-no-scroll`, plus add a following newline.
- **:append** Like `:append-no-scroll`, and scroll to the beginning of the new message.

**Compatibility note**
The values `:add-no-scroll` and `:add` for `where-keyword` are new in LispWorks 7.1.

**See also**
- `init-java-interface`
- `format-to-java-host`

**setup-deliver-dynamic-library-for-java**

**Function**

**Summary** Prepare for delivery of a dynamic library that can be loaded by Java.
Package lw-ji

Signature setup-deliver-dynamic-library-for-java &key init-java function asynchronous => result

Arguments

- init-java Boolean, default t.
- function A function designator for a function of no arguments.
- asynchronous Boolean, default nil.

Values result Always t.

Description The function setup-deliver-dynamic-library-for-java prepares the LispWorks internal state for delivering (using deliver) a dynamic library that can be loaded from Java and use the Java interface.

setup-deliver-dynamic-library-for-java should be called just before calling deliver. It causes the call to deliver to produce a dynamic library even if the call to deliver is not given any keywords that indicate it should produce a dynamic library. When the delivered image is loaded into Java by System.loadLibrary or System.load (or the underlying Runtime methods), the host Java virtual machine is remembered by LispWorks and can be retrieved by calling get-host-java-virtual-machine, and if init-java is non-nil (the default) the Java interface is automatically initialized by a call to init-java-interface. Finally, if function is non-nil, it is then called with no arguments.

The initialization of the Java interface and calling of function are done after the rest of the initialization of LispWorks. In particular, they occur after the deliver startup function (the first argument of deliver) has returned. If the deliver startup function does not return, the Java initialization does not occur.
asynchronous controls whether the initialization is asynchronous from the Java point of view. When asynchronous is false (the default), the Java method that loads Lisp waits until Lisp has finished initialization, initialized the Java interface and function (if non-nil) has been called and returned. When asynchronous is true, the loading method returns immediately and LispWorks initializes asynchronously.

In the asynchronous case, calls from Java to Lisp (using methods in the com.lispworks.LispCalls class) may happen before Lisp is ready. Such calls are blocked until Lisp is ready, or 50 seconds have passed. If Lisp is not ready within 50 seconds, you will get an exception. It is possible to check if Lisp is ready by using com.lispworks.LispCalls.waitForInitialization.

There is a minimal example of delivering LispWorks for Java in:

(example-edit-file "java/lisp-as-dll/README.txt")

Notes

function is intended for performing Java-specific initialization after the Java interface has been initialized, including any initialization that requires the Java interface (which therefore cannot be done by the deliver startup function).

setup-deliver-dynamic-library-for-java works by setting up and exporting the C symbol JNI_OnLoad, which the loading methods in Java invoke. If you want to export your own JNI_OnLoad, you must not use setup-deliver-dynamic-library-for-java.

The function get-host-java-virtual-machine can be used to get the host Java virtual machine. In the synchronous case (the default), get-host-java-virtual-machine returns nil when called from the deliver startup function (first argument to deliver), because Lisp did not receive the Java virtual machine yet. In the asynchronous case, get-host-java-virtual-machine returns the virtual machine from the beginning.
A non-Java program can also load a dynamic library that was created by delivering with `setup-deliver-dynamic-library-for-java`. In this case, the Java interface is not initialized automatically, `function` is not called, and `get-host-java-virtual-machine` returns `nil`. `get-host-java-virtual-machine` can be used as predicate to tell whether the loading was done from a Java program or not.

If you want initialization that happens only when loaded by Java but not otherwise and you need this to happen before the Java interface is initialized, the easiest approach is to pass `nil` for `init-java` and pass, as `function`, a function that does your initialization and before calling `init-java-interface`. For example:

```lisp
(setup-deliver-dynamic-library-for-java
 :init-java nil
 :function #'(lambda ()
               (my-java-pre-inits)
               (init-java-interface)
               (my-java-post-inits)))
```

If `init-java` is `nil`, you cannot use the Java interface until you call `init-java-interface`, as in the example above. The call to `init-java-interface` in this case can happen much later than the initialization, but note that calls from Java to Lisp that happen without checking if Lisp is ready will hang, and get an exception after 50 seconds.

In the asynchronous case with `nil` for `init-java`, `init-java-interface` can be called from the deliver startup function or later.

`setup-deliver-dynamic-library-for-java` modifies the way that `init-java-interface` looks for the virtual machine. In particular, you can call `init-java-interface` without specifying the Java virtual machine as in the example above, and `init-java-interface` will use `get-host-java-virtual-machine` to find it.
When `init-java` is true (the default), you can use `setup-java-interface-callbacks` to set some of the callbacks that in other situations you would pass to `init-java-interface`. `setup-java-interface-callbacks` should be called inside `function`, so they happen after the initialization of the Java interface.

LispWorks dynamic libraries that were delivered without using `setup-deliver-dynamic-library-for-java` can be loaded into Java, but to use the Java interface `init-java-interface` must be called with the Java virtual machine. It requires some expertise to pass the virtual machine to Lisp.

See also
- `init-java-interface`
- `get-host-java-virtual-machine`
- `setup-java-interface-callbacks`
- “Loading a LispWorks dynamic library into Java” on page 217

### setup-field-accessor

**Function**

**Summary** Defines a Java field accessor.

**Package** `lw-ji`

**Signature**

```lisp
setup-field-accessor name class-name field-name static-p
&optional is-final => result, error
```

**Arguments**

- `name` A symbol.
- `class-name` A string.
- `field-name` A string.
- `static-p` A boolean.
- `is-final` A boolean.

**Values**

- `result` `name` or `nil`.
error A condition or nil.

Description

The function `setup-field-accessor` defines a field accessor for a field in a Java class.

class-name must name a Java class.

field-name must be a field name.

static-p specifies whether the field is static or not.

is-final specifies whether the field is final (read-only) or not.

`setup-field-accessor` sets the symbol function of name to a function that reads the value of the field. If is-final is nil, it also defines (setf name) as the setter.

The arguments for the getter and setter are determined by the value of static-p. If static-p is non-nil, the getter takes no arguments and the setter takes the new value. If static-p is nil, the getter takes that object from which to get the value, and the setter gets the value and the object.

`setup-field-accessor` looks up the field definition in Java, and if the definition is incorrect returns nil and the condition as the second value.

Notes

In general, accessing fields should be avoided, because they are typically a less well-defined and implemented interface than methods, but sometimes it is necessary.

See also define-field-accessor

setup-java-caller

setup-java-constructor

Functions

Summary

Define a Java caller, which is a function that calls a Java method or a constructor.

Package lw-ji
The functions `setup-java-caller` and `setup-java-constructor` define a Java caller, which is a function that calls a Java method or a constructor. Once this the caller is defined, calls to `name` ultimately invoke the Java method or constructor.

Interpretation of the arguments and behavior of the defined caller is the same as the macros `define-java-caller` and `define-java-constructor`.

Unlike the macros `define-java-caller` and `define-java-constructor`, the functions `setup-java-caller` and `setup-java-constructor` do the lookup immediately, and therefore require running Java. If the lookup fails, they do not set the symbol function, and return two values: `nil` and a condition indicating the reason for the failure.

The functions (when successful) return `name`.

See also `define-java-caller`

“Defining specific callers” on page 202
setup-lisp-proxy  

Function

**Summary**  
Defines a Lisp proxy

**Package**  
lw-ji

**Signature**  
`setup-lisp-proxy name interface-and-method-descs => lisp-proxy-name`

**Arguments**  
`name` A symbol.
`interface-and-method-descs` A list.

**Values**  
`lisp-proxy-name` A symbol.

**Description**  
The function `setup-lisp-proxy` defines a Lisp proxy, as described for `define-lisp-proxy`.

Unlike `define-lisp-proxy name` can be `nil`, in which case `setup-lisp-proxy` generates a symbol by `cl:gensym`, uses it as the name and returns it.

`interface-and-method-descs` describes the Java interfaces to implement and the Lisp functions to call. `setup-lisp-proxy` takes it as a single argument, which must be a list, but otherwise parses it just like `define-lisp-proxy`.

**See also**  
`define-lisp-proxy`

---

standard-java-object  

Class

**Summary**  
A class for `jobject`.

**Package**  
lw-ji

**Superclasses**  
None.
Subclasses  None.

Initargs  :object

Accessors  java-instance-object

Description  The class standard-java-object is a class for object. Instances of standard-java-object can be passed to the Java interface functions and callers you define, and returned from Java calls whenever a object is needed. Each instance is normally associated with a object, which is used by the Java interface.

Apart from accessing the object in the instance, the Java interface does not do anything with the class, and makes no assumptions about it. There is no need for the class hierarchy in Lisp to reflect the class hierarchy in Java.

You can define your own classes that inherit from standard-java-object as well as other classes using standard def-class. Alternatively, you can tell the importing interface to define classes. There is no obvious advantage for using the latter.

The object slot defaults to nil, which the Java interface interprets as an invalid value. Your code needs to do something to set it. One option is to set it explicitly using the accessor java-instance-object. When you do that, the object must be a object, but the interface does not put any other restrictions. As long as it fits with the logic of your program, an instance of any Lisp class can hold a object of any Java class.

The other way to set the object slot is to use one of the interface functions that does it implicitly. This include the functions make-java-instance, create-instance-object and create-instance-object-list, and the :construct keyword argument to make-instance.
The :construct keyword is processed by an :after method of cl:initialize-instance on standard-java-object. When :construct is supplied, it needs to be either a list (possibly nil) of arguments for the constructor, or t, which means use default-constructor-arguments to get the argument list. The method then calls create-instance-jobject-list with the instance and the arguments to set create and set the jobject.

Additionally, the instance that is returned by create-instance-from-jobject also has the jobject.

Notes

When you pass :construct t, the call to default-constructor-arguments happens inside cl:initialize-instance, before all the cl:initialize-instance methods were called (the actual order of calls is the standard order). That means that if default-constructor-arguments depends on some values in the instance that may be set by an cl:initialize-instance method of another class, it may not work properly. You can avoid this problem by not passing the keyword :construct and instead using create-instance-jobject-list on the result of make-instance.

The interface for setting the jobject implicitly requires an association from the CLOS class name to the constructor, by using define-java-constructor or setup-java-constructor with the class-symbol argument.

create-instance-from-jobject requires an association from the Java class name to the CLOS class name, which is created by record-java-class-lisp-symbol.

throw-an-exception

Function

Summary

Throws a Java exception from a proxy method invocation.

Package

lw-ji
**Signature**

throw-an-exception  exception-class-or-exception  &rest  args

**Arguments**

exception-class-or-exception

A string or a Java object.

args

Lisp objects.

**Description**

The function throw-an-exception throws a Java exception from a proxy method invocation.

throw-an-exception must be called inside the function that is invoked from a proxy, otherwise a Lisp error is signaled. It causes throwing of a Java exception from the call.

exception-class-or-exception can be either a string naming an exception class or a Java object of an exception class. When it is a string, throw-an-exception constructs an exception of this class using args as arguments (same as create-java-object). If exception-class-or-exception is an exception then the args are ignored.

throw-an-exception throws in the Lisp sense out of your code, thus executing unwinding forms of surrounding cl:unwind-protect, and only then actually does the Java throwing (using JNI).

throw-an-exception can be used with the exception that is caught by catching-java-exceptions, if it is desired that the exception will be handled by the Java caller to the proxy. It is also needed when the method is documented to throw a specific exception in some situation.

**See also**

define-lisp-proxy

“Using proxies” on page 207

---

**to-java-host-stream***

**Variable**

**Summary**

An output stream that sends its output to the Java host.
Package lw-ji

Initial Value An output stream.

Description The variable \texttt{*to-java-host-stream*} is bound globally to an output stream that sends any output that is written to it to the Java host, by calling \texttt{send-message-to-java-host}. The \texttt{where-keyword} argument to \texttt{send-message-to-java-host} is \texttt{:add}, so the output is added at the end and scrolled if needed. If you do not want scrolling, you can use \texttt{*to-java-host-stream-no-scroll*} instead.

See \texttt{send-message-to-java-host} for details.

Notes The connection to the Java host is made by \texttt{init-java-interface}. Until \texttt{init-java-interface} has been called, output to \texttt{*to-java-host-stream*} does nothing. \texttt{*to-java-host-stream*} is not buffered and makes frequent calls to \texttt{send-message-to-java-host}. This should be OK for dealing with a few kilobytes for each user gesture.

See also \texttt{send-message-to-java-host} \texttt{*to-java-host-stream-no-scroll*}

\texttt{*to-java-host-stream-no-scroll*} Variable

Summary An output stream that sends its output to the Java host without scrolling.

Package lw-ji

Initial Value An output stream.

Description The variable \texttt{*to-java-host-stream-no-scroll*} is bound globally to an output stream that sends any output that is written to it to the Java host, by calling \texttt{send-message-to-}
java-host. The where-keyword argument to send-message-to-java-host is :add-no-scroll, so the output is added at the end, without ever scrolling. If you want it to scroll when needed, you can use *to-java-host-stream* instead.

See send-message-to-java-host for details.

Notes

The connection to the Java host is made by init-java-interface. Until init-java-interface is called, output to *to-java-host-stream-no-scroll* does nothing.

*to-java-host-stream-no-scroll* is not buffered and makes frequent calls to send-message-to-java-host. This should be OK for dealing with a few kilobytes for each user gesture.

See also send-message-to-java-host *to-java-host-stream*

### verify-java-caller

Function

#### Summary

Verify the Java caller.

#### Package

lw-ji

#### Signature

verify-java-caller name => result

#### Arguments

name A symbol.

#### Values

result A boolean.

#### Description

The function verify-java-caller verifies the Java caller, which means looking up the corresponding Java methods and setting up the caller for name.

name must be a caller name defined by define-java-caller (but not any of the other definers or setup-java-caller). If it is not, an error is signaled. Note that the importing inter-
face defines the caller using define-java-caller and that define-java-callers also expands to define-java-caller, so verify-java-caller can be used on such caller (but not on constructors or field accessors).

verify-java-caller looks up the Java class and method of the caller (unless they are already cached), and caches the information (so future calls to name or verification can use it).

verify-java-caller returns t if successful, nil otherwise.

verify-java-caller requires running Java.

Notes
1. In most cases using verify-java-callers to verify all the callers is more convenient.
2. Verification is useful to guard against typing mistakes when you typed the define-java-caller explicitly because that does not do any lookup until run time, or when you are not sure that the class definition has not changed between the time you imported the definition and the time it is used.

See also
verify-java-callers
define-java-caller
define-java-callers
“Calling from Lisp to Java” on page 200

**verify-java-callers**

*Function*

**Summary**
Verify all Java callers and return information about which was successful.

**Package**
lw-ji

**Signature**
verify-java-callers &key classes return => result

**Arguments**
classes A list of strings, or nil.
return \( t \), or one of the keywords \( \text{:name-only, :name, :info-only and :successful} \).

Values

result A list.

Description

The function \( \text{verify-java-callers} \) verifies all Java callers and returns information about which was successful and which was not.

If \( \text{classes} \) is non-nil, it must be a list of strings specifying Java classes. In this case, \( \text{verify-java-callers} \) verifies only callers for these classes. By default \( \text{verify-java-callers} \) verifies all callers that were defined by \( \text{define-java-caller} \).

\( \text{return} \) specifies what to return. See below for details.

\( \text{verify-java-callers} \) maps through all the callers that were defined by \( \text{define-java-caller} \) on all classes (if \( \text{classes} \) is \text{nil}) or on the supplied \( \text{classes} \).

Note that the importing interface defines the caller using \( \text{define-java-caller} \) and that \( \text{define-java-callers} \) also expands to \( \text{define-java-caller} \), so \( \text{verify-java-callers} \) verifies these callers too. \( \text{verify-java-callers} \) does not verify constructors or field accessors.

For each caller, \( \text{verify-java-callers} \) looks up the Java class and the method of the caller (unless it is already cached), and caches the information so calls to the caller and future verifications can use it.

\( \text{verify-java-callers} \) returns a list containing an item for each failed lookup, except when \( \text{return} \) is the keyword \( \text{:successful} \), in which case there is an item for each successful lookup. The value of each item depends on the value of \( \text{return} \) as follows:

\( t \) Each item is a cons \((\text{args . condition})\) where \( \text{args} \) is a list \((\text{name class-name method-name})\) of the required arguments of the \( \text{define-java-caller} \) form, and \( \text{condition} \) is the con-
dition that was produced when looking up. Unless something very unusual happened, this condition will be of type either \texttt{java-class-error} (if it failed to find the class) or \texttt{java-method-error} (if it failed to find the method).

\texttt{:name-only} Each item is the name of the caller that failed.

\texttt{:name} Each item is a cons where the \texttt{cl:car} is the name caller and the \texttt{cl:cdr} is the condition that was generated when trying the lookup.

\texttt{:info-only} Each item is the list \texttt{(name class-name method-name)} of the required arguments for \texttt{define-java-caller} of the failed caller.

\texttt{:successful} Each item is the name of a successful caller.

The default value of \texttt{return} is \texttt{t}.

\texttt{verify-java-callers} requires running Java.

Verification is useful to guard against typing mistakes when you typed the \texttt{define-java-callers} explicitly because that does not do any lookup until run time, or when you are not sure that the class definition has not changed between the time you imported the definition and the time it is used.

The intention is that you call \texttt{verify-java-callers} on starting your application, at least during the development phase, log the result and check it to see if anything is missing.

See also

\texttt{verify-java-caller}
\texttt{define-java-caller}
\texttt{define-java-callers}

“Calling from Lisp to Java” on page 200
verify-lisp-proxy
verify-lisp-proxies

Summary
Verify proxy definition(s).

Package
lw-ji

Signature
verify-lisp-proxy &optional do-undefined-method => unbounds, undefined-methods

Arguments
do-undefined-method
A generalized boolean.

Values
unbounds A list of lists, each of length 2.
undefined-methods A string or a list or nil.
defs-with-unbounds A list of lists.
defs-with-undefined-methods A list of lists.

Description
The function verify-lisp-proxy verifies a single proxy definition.

The function verify-lisp-proxies verifies all the proxy definitions that were defined by define-lisp-proxy (but not those created by setup-lisp-proxy).

Verify means two things:

- Check that all symbols in the definition which are not keywords have function definitions, and
- Check that the methods that are declared in the interfaces that the definition uses have method-specs. This check is
performed only if \texttt{do-undefined-method} is non-nil, and requires running Java. The default value of \texttt{do-undefined-method} is \texttt{nil}.

\texttt{verify-lisp-proxy} returns two values:

\textit{unbounds} reports symbols lacking function definitions. For each list in \textit{unbounds}, its first element is the method name, and its second element is the symbol that is not fbound. If the default function is not fbound, there is a list where the first element is "Default function".

\textit{undefined-methods} (if \texttt{do-undefined-method} is non-nil) can be either a string if one of the interfaces cannot be found (the string says that it cannot find an interface and gives its names), or a list. Each element in the list corresponds to an interface. The first element is the interface name, and the rest of the elements are strings specifying methods for which there is no matching \textit{method-descs}.

\texttt{verify-lisp-proxies} maps through the proxy definitions that were defined by \texttt{define-lisp-proxy}, and verifies each one of them. It returns two values, a list for definitions with symbols are not fbound, and a list for definitions with methods that are undefined. Each item of \textit{defs-with-unbounds} is a list corresponding to a definition with symbol not fbound, where the \texttt{cl:car} is the definition name and the \texttt{cl:cdr} is the \textit{unbounds} list as returned by \texttt{verify-lisp-proxy}. Each item in \textit{defs-with-undefined-methods} is a cons corresponding to a definition where a method is undefined, where the \texttt{cl:car} is the definition name and the \texttt{cl:cdr} is a string or a list of undefined methods as described above.

\textbf{Notes}

Failure to find an interface is a real error, and will cause \texttt{make-lisp-proxy} to signal error when trying to make a proxy. Symbols which are not fbound and missing methods would cause the default function to be called, which may or may not be the intention. Symbols that are not fbound are useful when they are intended to be always overridden, in
which case they should be keywords, so verification ignores them.

See also define-lisp-proxy

**write-java-class-definitions-to-file**
**write-java-class-definitions-to-stream**

*Functions*

**Summary**
Generate and output the definitions for a specified Java class.

**Package**
lw-ji

**Signature**

```
write-java-class-definitions-to-file java-class-name filename &key lisp-name lisp-class-p package-name prefix name-constructor export-p create-defpackage lisp-supers add-in-package print-case if-exists => java-class-name
```

```
write-java-class-definitions-to-stream java-class-name stream &key lisp-name lisp-class-p package-name name-constructor prefix export-p create-defpackage lisp-supers add-in-package print-case => java-class-name
```

**Arguments**

- `java-class-name` A string.
- `filename` A pathname designator.
- `stream` An output stream.
- `lisp-name` A symbol.
- `lisp-class-p` A generalized boolean.
- `package-name` A package designator.
- `prefix` A string or `nil`.
- `name-constructor` A function designator.
- `export-p` A generalized boolean.
- `create-defpackage` A generalized boolean.
- `lisp-supers` A list of symbols.
add-in-package  A generalized boolean.

print-case  One of the symbols :upcase, :downcase, or :capitalize.


Values  

java-class-name  A string.

Description  The functions write-java-class-definitions-to-file and write-java-class-definitions-to-stream generate the definitions for the Java class named by java-class-name, and then write them to the destination specified by filename or stream.

The generation of forms as the same as generate-java-class-definitions does, except that when add-in-package is non-nil write-java-class-definitions-to-stream and write-java-class-definitions-to-file insert a cl:in-package form after the package manipulation forms. The default value of add-in-package is non-nil.

The arguments java-class-name, lisp-name, lisp-class-p, package-name, name-constructor, prefix, export-p, create-defpackage and lisp-supers are processed as described in the entry for generate-java-class-definitions.

If add-in-package is non-nil, then after writing the package manipulation forms, a cl:in-package form is written with the package in which the definition names are interned, and the current package is bound to this package, which means the definition names do not need to be qualified with the package name.

print-case controls the binding of cl:*print-case* while outputting. The default value of print-case is :downcase.

if-exists is used by write-java-class-definitions-to-file when opening the file, in the same way as open.
write-java-class-definitions-to-stream generates the definitions for the class, and then writes all the definitions to the stream stream, with all the printer control variable set to the default except cl:*print-case* which takes its value from the print-case argument. It adds some comments, as lines starting with ";;;".

write-java-class-definitions-to-file first open the file for output using the filename and if-exists arguments, and then calls write-java-class-definitions-to-stream with all the arguments except filename and if-exists.

write-java-class-definitions-to-stream and write-java-class-definitions-to-file return the java-class-name.

Notes

1. write-java-class-definitions-to-stream and write-java-class-definitions-to-file require Java running, that is a working Java Virtual Machine and access to the definition of the class.

The generated code, however, is plain lisp, and can be compiled and loaded without Java. They allow you to use either of these functions once to generate the definitions, and add the output or file to your sources, and hence be able to compile and sources without running Java. Note that the output has no machine dependency at all. so as long as you can assume that the definition of the class does not change, you can output the definitions anywhere. For "globally public" classes (in the Java or Android packages), you can probably ask Lisp Support to generate the classes you need, and never bother with running Java on your development machine.

2. The output of these functions is all "user code", that is it uses only exported functions and macros that are available to user. It can be edited as desired, and definitions from it can be copied and used elsewhere.
3. **write-java-class-definitions-to-stream** is intended to allow writing the definitions of several classes to the same file. This especially useful when you write the definitions of several Java classes with the same package.

See also

- **generate-java-class-definitions**
- **import-java-class-definitions**
- “Importing classes” on page 201
The LW-JI Package
This chapter describes the Java classes and methods available in LispWorks. For an overview of this functionality with examples of use, see Chapter 15, “Java interface”.

com.lispworks.LispCalls

Java class

Summary

public class com.lispworks.LispCalls implements InvocationHandler

The Java class com.lispworks.LispCalls defines methods for calling from Java to Lisp.

com.lispworks.LispCalls is part of the LispWorks distribution. For Android it is part of the 7-1-0-0/etc/lisp-works.jar file. See the Chapter 16, “Android interface” for details. On other platforms it is defined in the JAR file lisp-calls.jar which is part of the LispWorks distribution in the etc directory, that is (lispworks-file "etc/lisp-calls.jar"). This JAR file needs to be on the classpath (for example by the keyword argument :java-class-path to init-java-interface).
com.lispworks.LispCalls.callIntV
com.lispworks.LispCalls.callIntA
com.lispworks.LispCalls.callDoubleV
com.lispworks.LispCalls.callDoubleA
com.lispworks.LispCalls.callObjectV
com.lispworks.LispCalls.callObjectA
com.lispworks.LispCalls.callVoidV
com.lispworks.LispCalls.callVoidA

Methods

public static int callIntV (String name, Object... args)
public static int callIntA (String name, Object[] args)
public static double callDoubleV (String name, Object... args)
public static double callDoubleA (String name, Object[] args)
public static Object callObjectV (String name, Object... args)
public static Object callObjectA (String name, Object[] args)
public static void callVoidV (String name, Object... args)
public static void callVoidA (String name, Object[] args)

Description

In the method name, the type specifies the return type, and V or A specifies whether the arguments are supplied as Variable arguments or Array. Otherwise the pairs of V and A methods behave the same.

The name argument is a string specifying a Lisp symbol. The name is parsed by a simple parser as described for com.lispworks.LispCalls.checkLispSymbol (with fboundp = true).
If the symbol is not found or is not fbound, these methods throw a `RuntimeException` with a string giving the reason for failure.

If the symbol is found, it is applied to the arguments `args`. For each argument, if it is a primitive type or of a class corresponding to a primitive type or a string, it is converted to the corresponding Lisp value. Otherwise it is passed as a `job-ject`. See “Types and conversion between Lisp and Java” on page 198. The result of the call is converted to the return type of the method and returned from the method. The conversion of the result type allows any float to be returned as a double, but does not coerce between integers and floats. For the `Object` return value, the result must be either a Java object (`jobject` or an instance of `standard-java-object`), or a Lisp object that can be converted to a Java object. See “Types and conversion between Lisp and Java” on page 198.

The Lisp function is an ordinary Lisp function, but it needs to return the right value. Unless the call is using the `Void` callers (`com.lispworks.LispCalls.callVoidA` or `com.lispworks.LispCalls.callVoidV`), returning the wrong value will call the `java-to-lisp-debugger-hook` (see `init-java-interface`) with an appropriate condition, and then return zero of the correct type (that is 0, 0d0 or Java `null`) from the call.

The call to the Lisp function is wrapped such that trying to throw out of it does not actually finish the throw, and instead returns zero of the correct type from the call. It is also wrapped by a debugger hook, which is invoked if the code tries to enter the debugger (normally as a result of an unhandled error, but could be any call to `cl:invoke-debugger`). The hook calls the `java-to-lisp-debugger-hook` (see `init-java-interface`) with the condition, and then calls `cl:abort`. If there is no `cl:abort` restart inside the Lisp function that catches this abort, this causes returning a zero of the correct type.
An important issue to remember is that when delivering with shaking, LispWorks eliminates symbols for which there is no reference. If the only call to a Lisp symbol foo is from Java, LispWorks will not see the reference and it will eliminate foo. To guard against this, you can either pass foo in a list to the deliver keyword :keep-symbols, or more conveniently, use the function hcl:deliver-keep-symbols (see the LispWorks Delivery User Guide), for example:

```lisp
(defun function-called-from-java (arg1 arg2)
  ...
)

(deliver-keep-symbols 'function-called-from-java)
```

Examples

```lisp
int sum = com.lispworks.LispCalls.callIntV (+", 2, 3, 10);
=> sum = 15

int position = com.lispworks.LispCalls.callIntV ("search", "r", "international");
=> position = 4

double logThree = com.lispworks.LispCalls.callDoubleV("log", 3);
=> logThree = 1.0986123
```

**com.lispworks.LispCalls.checkLispSymbol**

```java
public static boolean checkLispSymbol(String name, boolean fboundp)
```

**Description**

Checks whether a Lisp symbol exists, and optionally whether it is fbound.

`name` specifies the name of the Lisp symbol. The string `name` is parsed in a simple way, rather than using the Lisp reader. The parsing involves:

1. Upcase the string.
2. If there is a colon, take the part before it as a package name. Otherwise use "COMMON-LISP-USER" as the package name.

3. If the colon is followed by another colon, skip it and set a flag allowing internals. Otherwise, set a flag allowing only externals.

4. Take the rest of the string as the symbol name.

5. Find the package from the package name.

6. Find the symbol using the package and the symbol name. If it is internal, use it only if the flag allowing internal was set.

7. If \texttt{fboundp} is \texttt{true}, check whether the symbol is \texttt{fbound}. If all these steps succeed, \texttt{checkLispSymbol} returns \texttt{true}. Otherwise it returns \texttt{false}.

For symbols with names that do not need escaping, the result is the same normal processing by the Lisp reader without interning when there is no symbol.

\texttt{checkLispSymbol} caches the results in the Java side, which means that if the symbol appears or gets defined after the first call to \texttt{checkLispSymbol} it may return the wrong result.

\textbf{See also}\n
“Calling from Java to Lisp” on page 206
\texttt{init-java-interface}
\texttt{define-lisp-proxy}
\texttt{deliver}

\textbf{com.lispworks.LispCalls.createLispProxy}\n
\textit{Method}\n
\texttt{public static native Object createLispProxy(String name)}

\textbf{Description}\n
Creates a Lisp proxy, which is a Java proxy which calls Lisp functions.
name specifies a symbol which is the name of a proxy definition, defined in Lisp by either define-lisp-proxy or setup-lisp-proxy. The name is parsed by a simple parser as described for com.lispworks.LispCalls.checkLispSymbol (with fboundp = false).

Once it found the symbol, it makes a proxy the same way that calling make-lisp-proxy with name would, and returns it. The result is an Object that implements all the interfaces that are defined in the proxy definition, and when the methods of these interfaces are called on the object it calls into Lisp. See define-lisp-proxy for details.

If createLispProxy is successful it returns the proxy object. If there is any problem, this will cause a call to cl:error. If the cl:error call is not handled, the java-to-lisp-debugger-hook (see init-java-interface) is called with the condition, and then null is returned from createLispProxy. If the error is handled and tries to throw out of the context of the Lisp side of createLispProxy, the throw is blocked and createLispProxy returns null.

com.lispworks.LispCalls.waitForInitialization

Method

static public boolean waitForInitialization ()
static public boolean waitForInitialization (long seconds)
static public boolean waitForInitialization (long timeout , java.util.concurrent.TimeUnit unit)

Description Wait for a LispWorks dynamic library to finish initialization and accept foreign calls.

The method without arguments waits for up to 10000 seconds. The method that takes \texttt{long} waits for up to \texttt{seconds} seconds. The method that takes \texttt{long} and \\
\texttt{java.util.concurrent.TimeUnit} waits for up to the period defined by \texttt{timeout} and \texttt{unit}. See the Java documentation for the possible values of \texttt{java.util.concurrent.TimeUnit}.

\texttt{waitForInitialization} returns when LispWorks has finished its initialization or when the wait period has passed. If LispWorks finishes its initialization first, \texttt{waitForInitialization} returns true. If the wait period has passed, \texttt{waitForInitialization} returns false.

When \texttt{waitForInitialization} is called with 0 seconds, it returns immediately with true if LispWorks is already initialized, and false otherwise. Thus it can be used as a predicate without waiting.

Notes

Until LispWorks finishes its initialization, calls into LispWorks from Java using the other methods in \texttt{com.lispworks.LispCalls} hang, and raise an exception if hanging for too long. If this is an acceptable behavior, then you do not need \texttt{waitForInitialization}. If this is not acceptable, \texttt{waitForInitialization} allows you to check and avoid this situation. Typically your code will do something like:

\begin{verbatim}
if (com.lispworks.LispCalls.waitForInitialization(1))
    com.lispworks.LispCalls.callIntV("A-LISP-FUNCTION");
else
    do_something_else();
\end{verbatim}

If the LispWorks dynamic library was created with synchronous initialization (the default), then by the time the loading method (normally \texttt{System.loadLibrary} or \texttt{System.load}) returns, LispWorks has finished initializing. In this case you need \texttt{waitForInitialization} only in code that does not know if the loading method has returned (or even called at all).
If the LispWorks dynamic library was created with asynchronous initialization (*setup-deliver-dynamic-library-for-java* was called with true for *asynchronous*), the loading method returns immediately, and LispWorks initializes asynchronously. In this situation you can be sure that LispWorks finished initializing only after a call to *waitForInitialization* has returned true.

If you don’t know how the LispWorks dynamic library was created, just assume that it is asynchronous and always check using *waitForInitialization*.

**See also**  
*setup-deliver-dynamic-library-for-java*
This chapter describes the Android interface Java code.
For an overview of this functionality with examples of use, see Chapter 16, “Android interface”.

**com.lispworks.Manager**  
*Java class*

**Summary**

public class com.lispworks.Manager

A Java class that defines methods for using Lisp on Android. It contains one essential method, **com.lispworks.Manager.init**, which loads and initializes LispWorks. It also contains methods to set error reporters that will get called when an error inside Lisp is not caught by user handlers or when **report-error-to-java-host** is called, some methods to define where messages from Lisp (calls to **send-message-to-java-host** or **format-to-java-host**) go, and some other utilities.

**com.lispworks.Manager** defines these methods and fields:
Initialization

public static int init(Context context, String deliverName, Runnable reporter)

public static int init(Context context)

public static int init(Context context, Runnable reporter)

public static int init(Context context, String deliverName)

public static int status()

public static int init_result_code()

public static String mInitErrorString = ""

public static boolean loadLibrary()

public static boolean loadLibrary(String deliverName)

final public static int STATUS_INITIALIZING = 0

final public static int STATUS_READY = 1

final public static int STATUS_NOT_INITIALIZED = -1

final public static int STATUS_ERROR = -2

final public static int INIT_ERROR_NO_LIBRARY = -2000

final public static int INIT_ERROR_NO_ASSET = -2001

final public static int INIT_ERROR_FAIL_HEAP = -2002

Error handling

public static void setErrorReporter(LispErrorReporter ler)

public static void setGuiErrorReporter(LispGuiErrorReporter ler)

public interface LispErrorReporter

public interface LispGuiErrorReporter

public static synchronized void clearBugFormLogs(int count)
public static void showBugFormLogs(Activity act)

public static String mInitErrorString = ""

public static int mMaxErrorLogsNumber = 5

**Message handling**

public interface MessageHandler

public void setMessageHandler(MessageHandler handler)

public static synchronized void setTextView(android.widget.TextView textview)

public static void addMessage(String message, int where)

public static int mMessagesMaxLength

final public static int ADDMESSAGE_RESET = 0
final public static int ADDMESSAGE_APPEND = 1
final public static int ADDMESSAGE_PREPEND = 2
final public static int ADDMESSAGE_APPEND_NO_SCROLL = 3
final public static int ADDMESSAGE_ADD = 4
final public static int ADDMESSAGE_ADD_NO_SCROLL = 5

**Others**

public static void setCurrentActivity(android.app.Activity)

public static ClassLoader getClassLoader()

public static Context getApplicationContext()

**Notes**

The com.lispworks.Manager class is part of the LispWorks distribution, inside the lispworks.jar file.

**See also**

“Delivering for Android” on page 224

**com.lispworks.Manager.init**

Method

public static int init(Context context)

public static int init(Context context, Runnable reporter)
public static int init(Context context, String deliverName)

public static int init(Context context, String deliverName, Runnable reporter)

**Description**

Load and initialize Lispworks asynchronously.

`init` first checks whether LispWorks is already initialized or in the process of initializing, and if it is returns immediately the appropriate value (`STATUS_READY` or `STATUS_INITIALIZING`). Otherwise it loads LispWorks, and initiates the initialization process on another thread. It returns before initialization finished.

The argument `context` is any object of class `android.content.Context`. `init` uses it to find the application context, and hence where the LispWorks heap is.

The argument `reporter` is a `Runnable` that is invoked (that is its run method is invoked) when LispWorks finished initialization. The invocation is on the main thread. The `reporter` in general should use `com.lispworks.Manager.status` to check that initializing LispWorks succeeded. Once the `reporter` is invoked and `com.lispworks.Manager.status` returned `STATUS_READY`, it is possible to make calls into Lisp by methods in `com.lispworks.LispCalls`. If `reporter` is not supplied, it is possible to know that LispWorks is ready by two other mechanisms:

- Use the `com.lispworks.Manager.status` method from other places, or
- Call from Lisp into Java from the restart function (the first argument to `deliver-to-android-project`). When this restart function runs, LispWorks is already ready.

The argument `deliverName` specifies the name of the delivered LispWorks, specifically the base name of the heap and the dynamic library. See `deliver-to-android-project` for discussion. The default for `deliverName` is "LispWorks", which is
the default in deliver-to-android-project, so normally you do not need it.

`init` returns one of the `STATUS_*` constants. See the entry for `com.lispworks.Manager.status`.

`init` can be called repeatedly and it is thread-safe. The second and subsequent calls will not try to initialize it, unless the status is `STATUS_ERROR`, in which case it will try again. Each reporter that is passed to `init` is called independently. This is designed so if your application does not initialize LispWorks on startup, each part of it that relies on LispWorks can use `com.lispworks.Manager.status` to check whether LispWorks is ready, and if not call `init` with a reporter, and when the reporter is invoked check that `com.lispworks.Manager.status` returns `STATUS_READY`, and then rely on working LispWorks.

See also

`com.lispworks.Manager.status`
deliver-to-android-project
`com.lispworks.Manager`
“Delivering for Android” on page 224

`com.lispworks.Manager.status`  

```java
public static int status()

final public static int STATUS_INITIALIZING = 0
final public static int STATUS_READY =1
final public static int STATUS_NOT_INITIALIZED = -1
final public static int STATUS_ERROR = -2
```

**Description**

Return the status of LispWorks:

`STATUS_INITIALIZING`

LispWorks started initializing but has not finished yet. Because `com.lispworks.Manager.init` is asynchronous, it typically returns this value.

`STATUS_READY`

LispWorks finished initializing.

**STATUS_NOT_INITIALIZED**

LispWorks has not started initializing, that is before `com.lispworks.Manager.init` was called.

**STATUS_ERROR**

There was an error during initialization that prevented initialization. The method `com.lispworks.Manager.init_result_code` and the field `com.lispworks.Manager.mInitErrorString` gives more information about the reason for failure.

See also

- `com.lispworks.Manager.init`
- `com.lispworks.Manager.init_result_code`
- `com.lispworks.Manager.mInitErrorString`

"Delivering for Android" on page 224

### `com.lispworks.Manager.init_result_code` Method and Fields

```java
public static int init_result_code()
```

```java
final public static int INIT_ERROR_NO_LIBRARY = -2000
final public static int INIT_ERROR_NO_ASSET = -2001
final public static int INIT_ERROR_FAIL_HEAP = -2002
```

**Description**

Return a more detailed code specifying the result of the call to `com.lispworks.Manager.init`. The code is either one of the three `INIT_ERROR_*` constants above, or one of the codes that `InitLispWorks` returns.

**INIT_ERROR_NO_LIBRARY**

`com.lispworks.Manager.init` did not find the library.
Normally that would mean it is not in the project where it should be (\texttt{libs/armeabi-v7a} for Eclipse, \texttt{jniLibs/armeabi-v7a} for Android Studio), or its name is not correct. See \texttt{deliver-to-android-project} for details.

\textbf{INIT\_ERROR\_NO\_ASSET}

\texttt{com.lispworks.Manager.init} failed to find the LispWorks heap in the assets. Normally that means that the LispWorks heap is missing from the project (it should be in assets), or its name is incorrect. See \texttt{deliver-to-android-project} for details.

\textbf{INIT\_ERROR\_FAIL\_HEAP}

Extracting the heap from the assets failed. That in general should not happen. It may happen if the disk on the system is full.

Other values are documented for \texttt{InitLispWorks}. In general:

- 0 or greater means success (\texttt{com.lispworks.Manager.status} returns \texttt{STATUS\_READY}).
- Values greater than -100 and lower than 0 mean timeout. Since \texttt{com.lispworks.Manager.init} is asynchronous, that would be the values during initialization (\texttt{com.lispworks.Manager.status} returns \texttt{STATUS\_INITIALIZING}).
- -100 means not initialized (\texttt{com.lispworks.Manager.status} returns \texttt{STATUS\_NOT\_INITIALIZED}).
- Values lower than -100 indicate an error (\texttt{com.lispworks.Manager.status} returns \texttt{STATUS\_ERROR}).

\texttt{init\_result\_code} would typically be used after \texttt{com.lispworks.Manager.init} returned \texttt{STATUS\_ERROR}

When there is an error, \texttt{com.lispworks.Manager.mInitErrorString} contains a string describing it.
See also: com.lispworks.Manager.mInitErrorString
com.lispworks.Manager.init
com.lispworks.Manager.status
deliver-to-android-project
“Delivering for Android” on page 224

com.lispworks.Manager.mInitErrorString
Field

public static String mInitErrorString = ""

Description: Contains a string explaining the result for an error during initialization.

mInitErrorString is set to a non-empty string if there is an error during initialization of LispWorks, which would be detected either by using com.lispworks.Manager.status or com.lispworks.Manager.init_result_code.

The explanation is technical, so it will not be useful to show it to end users, but it should be helpful to developers, and certainly to LispWorks support.

See also: com.lispworks.Manager.init
com.lispworks.Manager.status
com.lispworks.Manager.init_result_code
“Delivering for Android” on page 224

com.lispworks.Manager.loadLibrary
Method

public static boolean loadLibrary()
public static boolean loadLibrary(String deliverName)

Description: Loads only the LispWorks dynamic library without initializing, for debugging.

Normally loadLibrary is called by com.lispworks.Manager.init, and in general you should not use it. It is supplied because it is sometimes useful for debugging.
**com.lispworks.Manager.init** can be called after **loadLibrary** was called, and will skip the call to it in this case.

*deliverName* has the same meaning as in **com.lispworks.Manager.init**.

Note that **loadLibrary** is not thread-safe on its own.

**loadLibrary** returns **true** on success, otherwise it returns **false** and sets **com.lispworks.Manager.mInitErrorString**.

See also **com.lispworks.Manager.init**
**com.lispworks.Manager.mInitErrorString**

**com.lispworks.Manager.LispErrorReporter**

**com.lispworks.Manager.setErrorReporter**

**com.lispworks.Manager.LispGuiErrorReporter**

**com.lispworks.Manager.setGuiErrorReporter**

### Methods and Interfaces

```java
public interface LispErrorReporter {
    boolean report(String errorString, String filename);
}
```

```java
public static void setErrorReporter(LispErrorReporter ler)
```

```java
public interface LispGuiErrorReporter {
    boolean report(String errorString, String filename);
}
```

```java
public static void setGuiErrorReporter(LispGuiErrorReporter ler)
```

### Description

Set error reporters that gets invoked when either **report-error-to-java-host** is called, or an error is not caught by your handler or hook.

**setErrorReporter** and **setGuiErrorReporter** are used to set error reporters. When either **report-error-to-java-host** is called (by your code, the system does not use it) or an error is not handled by your handlers (including debugger-
wrappers and cl:*debugger-hook*), the report method of the interface is invoked. By default the reporters are both null.

The errorString of the report message is a string describing the error. The filename is the name of a file that contains a log file, but can be also null.

**Note:** when report-error-to-java-host is called it is your responsibility to pass the right strings.

The reporters should do whatever you want to do. The return value should indicate if the error was dealt with completely, so there is no need to call com.lispworks.Manager.addMessage (see below).

The reporter that is set by setErrorReporter ("the Lisp error reporter") and the reporter that is set by setGuiErrorReporter ("the Lisp GUI error reporter") differ by the scope in which their report method is invoked:

- The report method of the Lisp error reporter is invoked within the scope of the error, which also means it can be any thread. It is therefore cannot do anything related to the GUI, and needs to be runnable on any thread. In general, it should only set internal variables and return, but it may also do things like copying the log file somewhere.

- The report method of the Lisp GUI error reporter is invoked outside the scope of the error, on the GUI thread. It is done by the event loop of the GUI thread, so it is also synchronous with respect to processing events. It can therefore safely access the GUI and perform what is needed to inform the user that an error has occurred.

setErrorReporter and setGuiErrorReporter can be called at any time, before or after com.lispworks.Manager.init. There is only one Lisp error reporter and one Lisp GUI error reporter, and each call to setErrorReporter
or `setGuiErrorReporter` overwrites the previous value. The reporters can be set to `null`.

When Lisp calls into Java to report an error, it does the following steps:

1. If `com.lispworks.Manager.mMaxErrorLogsNumber` is greater than 0, records the error and delete previous record(s) if the number of records reached `com.lispworks.Manager.mMaxErrorLogsNumber` (these records can be displayed by `com.lispworks.Manager.showBugFormLogs`).

2. If the Lisp error reporter (the non-GUI one) is not `null`, invoke its `report` method.

3. If the Lisp GUI error reporter is not `null`, arrange for its `report` method to be invoked on the GUI process, and does the next 2 steps after this invocation.


5. If `com.lispworks.Manager.mMaxErrorLogsNumber` is not greater than 0, delete the log file if it is not `null`.

Notes

The log files are deleted when LispWorks starts (when `com.lispworks.Manager.init` is successful). They are also in the internal cache directory, which means they are not visible to other applications. If you want to make the logs visible, the reporter needs to copy the file to an external directory.

See also

- `report-error-to-java-host`
- `com.lispworks.Manager.init`
- `com.lispworks.Manager.addMessage`
- `com.lispworks.Manager.showBugFormLogs`
- `com.lispworks.Manager.mMaxErrorLogsNumber`
**com.lispworks.Manager.clearBugFormLogs**

```java
public static synchronized void clearBugFormLogs(int count)
```

**Description**
Clear the bug form logs list.

LispWorks keeps a record of error reports containing the error strings and the file names containing the log (the arguments the report method of `com.lispworks.Manager.LispErrorReporter` received). `clearBugFormLogs` eliminates all entries except the last `count` entries, and removes the files.

The record is limited to `com.lispworks.Manager.mMaxErrorLogsNumber`, which defaults to 5.

The record can be displayed by `com.lispworks.Manager.showBugFormLogs`, which allows the user to open the log file of a record by selecting it.

**Notes**
The log files are also automatically deleted when LispWorks starts (that is when `com.lispworks.Manager.init` is successful).

**See also**
- `com.lispworks.Manager.setErrorReporter`
- `com.lispworks.Manager.mMaxErrorLogsNumber`
- `com.lispworks.Manager.showBugFormLogs`

**com.lispworks.Manager.mMaxErrorLogsNumber**

```java
public static int mMaxErrorLogsNumber = 5
```

**Description**
Maximum number of error logs to keep.

The default value of 5 is a compromise between keeping many logs (in case some are useful) and avoiding filling the disk. During development you may want to enlarge it, and in the finished product maybe reduce it, possibly to 0.

The log files are deleted when LispWorks is initialized.
com.lispworks.Manager.showBugFormLogs 

**Method**

```java
public static void showBugFormLogs(Activity act)
```

**Description**

This method is for debugging.

`showBugFormLogs` shows a list of the `BugFormLogs`, where each item is an error string, and allows you to open the associated log file by touching the item. If there is only one item, it opens it immediately.

The argument `act` is the activity that invokes the bug list.

The bug list is displayed in its own activity, `com.lispworks.BugFormLogsList`, and the log file is opened to another activity, `com.lispworks.BugFormViewer`. To make `showBugFormLogs` work, you must add these activities to the file `AndroidManifest.xml` in your project like this:

```
<activity android:name="com.lispworks.BugFormViewer"
    android:label="Bug Form viewer"> </activity>
<activity android:name="com.lispworks.BugFormLogsList"
    android:label="Bug Form Logs"> </activity>
```

The `AndroidManifest.xml` of the OthelloDemo examples contains these lines. Apart from putting the activities in the `AndroidManifest.xml`, you should not do anything else with them.

This method shows Lisp bug forms, so is useful only for Lisp developers.

There will not be any bug form logs if there was no error, or `com.lispworks.Manager.mMaxErrorLogsNumber` is set to 0, in which case `showBugFormLogs` does nothing. It is also possible for the user error reporters (see `com.lispworks.Manager.setErrorReporter`) to delete the log files, so `com.lispworks.BugFormViewer` will fail to show it.

`showBugFormLogs` is useful during development. Once the application is working, you probably want to remove the activities from `AndroidManifest.xml` and not use `showBugFormLogs`. 
See also  
com.lispworks.Manager.mMaxErrorLogsNumber  
com.lispworks.Manager.setErrorReporter  
com.lispworks.BugFormLogsList  
com.lispworks.BugFormViewer

**com.lispworks.Manager.addMessage**  
**com.lispworks.Manager.mMessagesMaxLength**  

Method and Fields

```java
public static void addMessage(String message, int where)
public static int mMessagesMaxLength = 10000
final public static int ADDMESSAGE_RESET = 0
final public static int ADDMESSAGE_APPEND = 1
final public static int ADDMESSAGE_PREPEND = 2
final public static int ADDMESSAGE_APPEND_NO_SCROLL = 3
final public static int ADDMESSAGE_ADD = 4
final public static int ADDMESSAGE_ADD_NO_SCROLL = 5
```

Description  
Adds a message.

The actual meaning of adding a message is either to call the message handler if it was set by `com.lispworks.Manager.setMessageHandler`, or put the message in the output text view if it was set by `com.lispworks.Manager.setTextView`, if neither the handler or the view are set, then `addMessage` accumulates the messages, and inserts the text next time that that `com.lispworks.Manager.setTextView` is called.

The operation of `addMessage` is first to check whether the handler is not null, and if it is call the handler with the two arguments. If the handler returns `true`, `addMessage` does not do anything else. Otherwise, if there is a textview it adds the message to it, otherwise it adds the message to its own buffer. `where` needs to be one of the six `ADDMESSAGE_*` constants, and determines how the message is added. `ADDMESSAGE_RESET` causes `addMessage` to first clear the textview or the internal string before adding the message. `ADDMESSAGE_PREPEND` means adding the string at the beginning of the textview or
internal string, followed by a newline. ADDMESSAGE_APPEND, ADDMESSAGE_APPEND_NO_SCROLL, ADDMESSAGE_ADD and ADDMESSAGE_ADD_NO_SCROLL all add the message to the end of the textview or internal string. ADDMESSAGE_APPEND and ADDMESSAGE_APPEND_NO_SCROLL follow the message by a newline, while the ADDMESSAGE_ADD and ADDMESSAGE_ADD_NO_SCROLL do not.

ADDMESSAGE_APPEND_NO_SCROLL and ADDMESSAGE_ADD_NO_SCROLL do not scroll, while ADDMESSAGE_APPEND and ADDMESSAGE_ADD scroll the textview to make at least the top of the new message visible.

addMessage is used by LispWorks to perform the operation of send-message-to-java-host, and to report errors which are not dealt with by the error reporters. You can use it when it is useful.

The call to the handler is done on the thread on which addMessage is called, so the handler must be able to cope with being called on any thread, and needs to be thread-safe. The access to the textview or the internal string is done on the GUI thread and is thread-safe.

mMessagesMaxLength limits the length that addMessage accumulates. The length of the text that addMessage accumulates, either internally or in the TextView, is limited to the value mMessagesMaxLength (default 10000). When appending causes the length to overflow this value, addMessage removes the beginning of the old accumulated text so the total is the limited to mMessagesMaxLength. However, it does not remove part of the message itself, so calling addMessage with a string longer than mMessagesMaxLength will cause the TextView or internal string to be longer than mMessagesMaxLength (the old text would be removed completely in this case).

Compatibility notes

ADDMESSAGE_ADD and ADDMESSAGE_ADD_NO_SCROLL are new in LispWorks 7.1. In LispWorks 7.0, the ADDMESSAGE_APPEND and ADDMESSAGE_APPEND_NO_SCROLL inserted the newline
before the message. If you rely on that, you may have to modify your code.

See also  
send-message-to-java-host  
com.lispworks.Manager.setErrorReporter  
com.lispworks.Manager.setMessageHandler  
com.lispworks.Manager.setTextView

**com.lispworks.Manager.setMessageHandler**  
**Method and Interface**

```java
public void setMessageHandler(MessageHandler handler) {
    mMessagehandler = handler;
}

public interface MessageHandler {
    boolean handle(String message, int where);
}
```

**Description**  
Sets the message handler which `com.lispworks.Manager.addMessage` uses.

The `handler` is `null` by default, and can be set to `null`.

When the `handler` is not `null`, `com.lispworks.Manager.addMessage` calls the `handle` method with its arguments. The result tells `com.lispworks.Manager.addMessage` whether to deal further with the string, see its reference entry for further details.

Note that the `handler` can be called on any thread, and needs to be thread-safe.

See also  
com.lispworks.Manager.addMessage

**com.lispworks.Manager.setTextView**  
**Method**

```java
public static synchronized void setTextView(android.widget.TextView textview)
```
Sets the textview for `com.lispworks.Manager.addMessage`.

The textview defaults to null and can be set to null. When it is null, `com.lispworks.Manager.addMessage` accumulates the message.

When `setTextView` is called, if there is already a textview it takes the content first and puts it in the buffer of `com.lispworks.Manager.addMessage`. If the new value textview is not null, it puts into it the buffer of `com.lispworks.Manager.addMessage` and clears the buffer. This is designed such that you can set the Textview to another Textview or to null without losing text.

The intention is that Textview makes it easy to display messages that come from Lisp. In a fully-developed product you probably want a better mechanism, by setting the message handler with `com.lispworks.Manager.setMessageHandler`.

There is no expectation by `setTextView` or `com.lispworks.Manager.addMessage` about the properties of the Textview except that it is possible to add text to it and delete all the text from it. You can manipulate it yourself (for example delete all the text, or all the text except the last 100 lines) while is set.

`setTextView` can be called on any thread, and is thread-safe.

See also:

- `com.lispworks.Manager.addMessage`
- `com.lispworks.Manager.setMessageHandler`

Methods

```java
public static ClassLoader getClassLoader()

public static Context getApplicationContext()
```
Description
Return the application context of the context that was supplied to com.lispworks.Manager.init, and the ClassLoader associated with it.

These are utility methods that LispWorks itself uses and you may find useful. They must be called only after com.lispworks.Manager.init was called.

See also
com.lispworks.Manager.init

com.lispworks.Manager.setCurrentActivity

Method

```java
public static void setCurrentActivity(android.app.Activity activity)
```

Description
Sets the current activity that can be used inside Lisp using android-get-current-activity.

The argument activity must be the current active Activity, or null. The Lisp function android-get-current-activity returns this activity.

Once the activity becomes inactive, setCurrentActivity needs to be called with null, or the new active Activity.

Notes
1. setCurrentActivity is effectively licensing the Lisp side to raise dialogs in the current activity.

2. Activity instances that are used in setCurrentActivity should reset it by calling it with null in their onPause method, to ensure that they are not used after they are no longer visible.

3. Activities that allow Lisp to raise dialogs throughout their lifetime should set it on in the onResume method.

4. If all the activities in the application set the current activity, then you do not need to reset it in the onPause method.
5. `setCurrentActivity` only affects what `android-get-current-activity` returns. Code that gets the `Activity` in other way will not be affected.

See also `android-get-current-activity`

**com.lispworks.BugFormLogsList**

*Java class*

```java
class BugFormLogsList extends ListActivity
```

**Description**

Used by `com.lispworks.Manager.showBugFormLogs` to show the list of bug form logs.

**See also** `com.lispworks.Manager.showBugFormLogs`

**com.lispworks.BugFormViewer**

*Java class*

```java
class BugFormViewer extends Activity
```

**Description**

Used by `com.lispworks.Manager.showBugFormLogs` to show an individual log.

**See also** `com.lispworks.Manager.showBugFormLogs`
Android Java classes and methods
This chapter describes symbols available in the MP package, giving you access to the multiprocessing capabilities of LispWorks.

Multiprocessing is discussed in detail in Chapter 19, “Multiprocessing”.

allowing-block-interrupts

Summary
Allows control over blocking interrupts.

Package
mp

Signature
allowing-block-interrupts start-blocked &body body => results

Arguments
start-blocked A generalized boolean
body Code

Values
results Values returned by evaluating body.

Description
The macro allowing-block-interrupts executes body allowing control over blocking interrupts by current-pro-
cess-unblock-interrupts and current-process-unblock-interrupts.

Within the dynamic scope of allowing-block-interrupts, you can switch the blocking of interrupts on and off. Blocking interrupts prevents any interruption of the current process, including process-interrupt, process-kill, process-reset, process-break and process-stop. These interrupts are all queued and processed once interrupts become unblocked.

Blocking interrupts also blocks interrupts due to POSIX signals. Such interrupts are processed either by another Lisp thread, or once interrupts become unblocked.

If start-blocked is true, allowing-block-interrupts blocks interrupts on entry. If start-blocked is false, the state does not change on entry. If you want to ensure that the initial forms of allowing-block-interrupts are interruptible even if it is inside the scope of another allowing-block-interrupts, you need to explicitly call current-process-unblock-interrupts on entry.

allowing-block-interrupts can be used recursively.

In compiled code, allowing-block-interrupts with a true value of the start-blocked argument is guaranteed not to process interrupts before an explicit change to the blocking state (that includes exiting the scope of allowing-block-interrupts). In particular, if the first cleanup form of an unwind-protect is a call to allowing-block-interrupts, it is guaranteed to execute without interrupts on exit from the protected form. No such guarantee is given in interpreted code.

On exit from allowing-block-interrupts, the current state of interrupt blocking and whether there is a surrounding use of allowing-block-interrupts or with-interrupts-blocked is restored to the state that existed on entry.

allowing-block-interrupts returns the results of body.
any-other-process-non-internal-server-p

Function

Summary
Tests whether there is any other process except the caller that is not marked as "internal server".

Package
mp

Signature
any-other-process-non-internal-server-p => result

Arguments
None.

Values
result A boolean.

Description
The function any-other-process-non-internal-server-p is the predicate for whether there is any other process, except the caller process, that is not marked as "internal server".

Notes
Processes are marked as "internal server" by a true value for :internal-server amongst the keywords in a call to process-run-function.

See also
process-run-function
process-internal-server-p
barrier

System Class

Summary
A class of objects for synchronizing processes.

Package mp

Superclasses t

Description
Instances of the system class barrier are used for synchronizing processes. They are made by make-barrier and barrier-wait is typically called at synchronization points.

See also
make-barrier
barrier-wait
“Synchronization barriers” on page 287

barrier-arriver-count

Function

Summary
Returns the arriver count of a barrier.

Package mp

Signature
barrier-arriver-count barrier => result

Arguments
barrier  A barrier.

Values
result  A positive fixnum, or nil.

Description
The function barrier-arriver-count returns the arriver count of the barrier barrier, or nil for a disabled barrier.

Notes
If barrier is in use, the arriver count can change at any time.

See also
barrier
barrier-wait
**make-barrier**

“Synchronization barriers” on page 287

---

**barrier-block-and-wait**

**Function**

**Summary** Enables a barrier, waits until a specified number of arrivers arrive, and then wakes immediately.

**Package** mp

**Signature**

```lisp
barrier-block-and-wait barrier count &key wait-if-used-p errorp timeout unblock => result
```

**Arguments**

- `barrier` A barrier.
- `count` A positive integer.
- `wait-if-used-p` A generalized boolean.
- `errorp` A boolean.
- `timeout` A non-negative real or nil.
- `unblock` A boolean.

**Values**

- `result` An integer, a symbol or a mp:process object.

**Description**

The function `barrier-block-and-wait` enables the barrier `barrier` with `t`, that is it makes any number of arrivers wait, and then waits until `count` arrivers arrive.

`wait-if-used-p` controls whether to wait if another process is already inside `barrier-block-and-wait`. The default value of `wait-if-used-p` is `nil`.

`barrier` is a barrier made by `make-barrier`.

`errorp` controls whether to signal an error if another process is already inside `barrier-block-and-wait` and `wait-if-used-p` is `nil`. The default value of `errorp` is `nil`. 

---

1363
timeout, if non-nil, specifies the time in seconds to wait before timing out. The default value of timeout is nil.

unblock specifies whether processes that already wait on barrier should be unblocked first. The default value of unblock is nil.

`barrier-block-and-wait` is "using" barrier, and only one process can do this the same time. `barrier-block-and-wait` first tries to mark barrier as used by the current process, which will fail if another process is inside `barrier-block-and-wait` with the same barrier. In this case it does one of three options:

1. If `wait-if-used-p` is non-nil, it calls `barrier-wait` on barrier (without any keyword argument) and returns the result.

2. If `errorp` is non-nil, it calls `error`.

3. Otherwise is returns the other process.

Once `barrier-block-and-wait` has successfully marked barrier as used, it changes its count to t as if by calling `(barrier-change-count barrier t)`, which will cause other `barrier-wait` calls to wait. If `unblock` is non-nil, it first unblocks all processes that wait on the barrier, so the effect is the same as `(barrier-enable barrier t)`.

It then waits until the arriver count of barrier is greater than or equal to count, or, if `timeout` is supplied, `timeout` seconds passed. It then returns the number of arrivers.

result can be one of three types:

- **integer** The call was successful, and result is the number of arrivers.
- **symbol** barrier was in use, and `wait-if-used-p` is non-nil, so `barrier-wait` was called. result is the result of `barrier-wait`.
- **mp:process** barrier is in use, and result is the process that uses it.
Notes

1. When `barrier-block-and-wait` returns, `barrier` is still set with `t`, that is calls to `barrier-wait` on barrier will wait. Normally the current process will go on to do some operations that require the other processes to wait, and then release them by calling `barrier-disable` or `barrier-enable`.

2. In typical usage, the arriver count is just increased by one by each call to `barrier-wait`, so as long as other processes use only `barrier-wait` (or `barrier-block-and-wait` with `wait-if-used-p` non-nil), `barrier-block-and-wait` will return after `count` processes called `barrier-wait` and are waiting. That is the intended purpose of `barrier-block-and-wait`. If other processes call functions that manipulate the arriver count or the count of `barrier` (`barrier-disable`, `barrier-enable`, `barrier-unblock`, `barrier-change-count`), then `barrier-block-and-wait` will "get confused", in the sense that while its behavior is still well-defined, it is not intuitive.

3. With the default keyword values (not counting `timeout`), `barrier-block-and-wait` is useful for controlling a fixed set of processes by another "master" process. The processes in the set need to call `barrier-wait` at appropriate points. When the "master" process wants to stop them for a while, it calls `barrier-block-and-wait`. When it wants to restart them, it calls `barrier-disable`.

4. A non-nil value of `wait-if-used-p` is useful when any member of a group of processes may decide that it needs to stop all the other processes in the group. In this case, this process calls `barrier-block-and-wait` with `wait-if-used-p` non-nil (and count the number of processes in the group minus one). If two of the processes happen to call it at the same time, one will get the barrier, and the other process will have to wait.
5. The effect of `barrier-block-and-wait` can be approximated by using `barrier-change-count` followed by normal `process-wait` that checks the arrivers count in the wait function. `barrier-block-and-wait` has two advantages:

a) It checks against more than one process trying to do it at the same time.

b) `barrier-block-and-wait` will wake up immediately when the arriver count reaches the right number. `process-wait` will wake up only when the scheduler checks the wait function and wakes it up.

See also
- `barrier`
- `barrier-wait`
- `make-barrier`
- `barrier-enable`
- `barrier-disable`

“Synchronization barriers” on page 287

### `barrier-change-count` Function

**Summary**
Changes the count of a `barrier`.

**Package**
`mp`

**Signature**
`barrier-change-count barrier new-count => result`

**Arguments**
- `barrier` A `barrier`.
- `new-count` A positive fixnum, or `t` meaning `most-positive-fixnum`.

**Values**
- `result` A boolean.

**Description**
The function `barrier-change-count` changes the count of the `barrier` `barrier` to `new-count`. 
If barrier is enabled and the arriver count is less than new-count, this just sets the count of barrier to new-count and returns t. Otherwise, it calls

(barrier-unblock barrier :reset-count new-count)

and returns nil.

See also

barrier
barrier-unblock
“Synchronization barriers” on page 287

**barrier-count**

*Function*

**Summary**

Returns the current count of a barrier.

**Package**

mp

**Signature**

barrier-count barrier => result

**Arguments**

barrier A barrier.

**Values**

result A positive fixnum, or nil.

**Description**

The function barrier-count returns the current count of the barrier barrier, or nil if barrier is disabled.

**Notes**

The count value can be changed by barrier-unblock, barrier-enable, barrier-disable or barrier-change-count.

**See also**

barrier
barrier-wait
make-barrier
barrier-change-count
barrier-disable
barrier-enable
barrier-unblock

“Synchronization barriers” on page 287

### barrier-disable

**Function**

**Summary**
Unblocks and disables a `barrier`.

**Package**
`mp`

**Signature**

```lisp
barrier-disable barrier &optional kill-waiting
```

**Arguments**

- `barrier` A `barrier`.
- `kill-waiting` A boolean.

**Description**

The function `barrier-disable` unblocks the `barrier` and then disables it. If `kill-waiting` is true, `barrier-disable` also kills any waiting thread. This is done by calling

```lisp
(bARRIER-unblock barrier :disable t :kill-waiting kill-waiting)
```

**See also**

- `barrier`
- `barrier-unblock`
- `barrier-wait`
- `make-barrier`
- “Synchronization barriers” on page 287

### barrier-enable

**Function**

**Summary**
Ensures that a `barrier` is enabled.

**Package**
`mp`

**Signature**

```lisp
barrier-enable barrier count &optional kill-waiting
```
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>barrier</td>
<td>A barrier.</td>
</tr>
<tr>
<td>count</td>
<td>A positive fixnum, or t meaning most-positive-fixnum.</td>
</tr>
<tr>
<td>kill-waiting</td>
<td>A boolean.</td>
</tr>
</tbody>
</table>

Description

The function `barrier-enable` ensures that the `barrier` `barrier` is enabled after unblocking it if it is already enabled, and sets its count to `count`. If `kill-waiting` is true, `barrier-enable` also kills any waiting threads. This is done by calling

```
(barrier-unblock barrier :reset-count count :kill-waiting kill-waiting)
```

See also

- barrier
- barrier-wait
- make-barrier
- barrier-unblock
- “Synchronization barriers” on page 287

### barrier-name

**Function**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Returns the name of a <code>barrier</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td><code>mp</code></td>
</tr>
<tr>
<td>Signature</td>
<td><code>barrier-name barrier =&gt; name</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>barrier</code></td>
</tr>
<tr>
<td>Values</td>
<td><code>name</code></td>
</tr>
</tbody>
</table>

**Description**

The function `barrier-name` returns the name of `barrier`, as supplied or defaulted in the call to `make-barrier`. 
See also  
barrier  
make-barrier  
“Synchronization barriers” on page 287

**barrier-pass-through**  
*Function*

**Summary**  
Increments the arriver count of a **barrier**.

**Package**  
mp

**Signature**  
`barrier-pass-through barrier => result`

**Arguments**  
`barrier`  
A **barrier**.

**Values**  
`result`  
One of the keywords **:unblocked** and **:passed-through**.

**Description**  
The function **barrier-pass-through** increments the arriver count of the **barrier**. If the arriver count thereby reaches the count, **barrier-pass-through** unblocks **barrier** and returns **:unblocked**, otherwise it returns **:passed-through**.

**barrier-pass-through** is equivalent to calling **barrier-wait** with **pass-through t**. See **barrier-wait** for details.

See also  
barrier  
barrier-wait  
make-barrier  
“Synchronization barriers” on page 287

**barrier-unblock**  
*Function*

**Summary**  
Unblocks a **barrier**.
Package  mp

Signature  barrier-unblock  barrier &key  disable  reset-count  kill-waiting

Arguments  

barrier  A  barrier.
disable  A  boolean.
reset-count  A  positive  fixnum,  t  or  nil.
kill-waiting  A  boolean.

Description  The  function  barrier-unblock  unblocks  the  barrier  barrier,  potentially  disabling  it,  resetting  its  count  or  killing  the  waiting  processes.

Without  keyword  arguments,  barrier-unblock  unblocks  barrier,  which  means  that  any  thread  that  is  waiting  on  barrier  wakes  and  returns  from  barrier-wait,  and  its  arriver  count  is  reset  to  0.

If  disable  is  true,  or  if  disable  is  not  passed  and  barrier  was  made  with  disable-on-unblock  true,  then  barrier-unblock  also  disables  barrier,  so  any  further  calls  to  barrier-wait  return  nil  immediately.

If  reset-count  is  non-nil,  it  must  be  valid  count  (a  positive  fixnum  or  t),  and  barrier-unblock  sets  the  count  of  barrier  to  this  value.

If  kill-waiting  is  true,  instead  of  waking  up  the  waiting  threads,  barrier-unblock  kills  them  (by  process-terminate).

See  also  process-terminate  barrier  barrier-wait  make-barrier  “Synchronization  barriers”  on  page  287
**barrier-wait**  

*Function*

**Summary**  
Waits on a barrier until enough threads arrive.

**Package**  
*mp*

**Signature**  
`barrier-wait barrier &key timeout callback pass-through  
discount-on-abort discount-on-timeout disable-on-unblock => result`

**Arguments**  
- `barrier`: A barrier.
- `timeout`: A non-negative real or `nil`.
- `pass-through`: A boolean.
- `discount-on-abort`: A boolean.
- `discount-on-timeout`: A boolean.
- `disable-on-unblock`: A boolean.
- `callback`: A function designator.

**Values**  
- `result`: `t`, `nil` or one of the keywords `:unblocked`, `:passed-through` and `:timeout`.

**Description**  
The function `barrier-wait` waits on a barrier until enough threads arrive on `barrier`. When `barrier-wait` is called, it "arrives", and when the number of arrivers reaches the count of the barrier (that is, the `count` argument to `make-barrier`), `barrier-wait` returns. Effectively, the last "arriver" unblocks the barrier and wakes up all the other waiting threads.

`timeout` is the maximum time to wait in seconds.

If `pass-through` is true, `barrier-wait` performs the other operations but does not wait.

`discount-on-abort` controls whether to change the arrivers count of `barrier` on an abort (see later).
discount-on-timeout controls whether to change the arrivers count of barrier on a timeout (see later).

disable-on-unblock controls whether to disable barrier when unblocking.

callback, if supplied, specifies a callback called before unblocking.

barrier-wait first checks whether barrier is disabled, and if it is then barrier-wait returns nil immediately. It then checks the number of arrivers of barrier, which is the number of other calls to barrier-wait on the same barrier since it was last unblocked or created.

If the number of arrivers is less than the count minus 1, barrier-wait increases the number of arrivers by 1, and then waits for barrier to be unblocked (unless pass-through is true, which causes it to return immediately). If the number of arrivers of barrier equals its count minus 1, then barrier-wait unblocks barrier (as described below) and returns :unblocked.

discount-on-abort, discount-on-timeout, disable-on-unblock and callback allow you to control how barrier-wait waits and also how barrier is unblocking. For each of these, the effective value is either that supplied to barrier-wait, or if it was not supplied to barrier-wait, the value in barrier itself (see make-barrier).

timeout can be used to limit the time that barrier-wait waits. It is either a number of seconds or nil (the default), meaning wait forever. If barrier-wait times out, it returns :timeout. By default it does not change the number of arrivers after a timeout, so the call is still counted as an "arrival", but this can be changed by using discount-on-timeout. If discount-on-timeout is true then barrier-wait decrements the arrivers count after a timeout, so the call has no overall effect on the arrivers count.
If `barrier-wait` is aborted while it waits (for example by `process-terminate` or throwing using `process-interrupt`), it does not change the arrivers count by default, so the call still counts as an arrival, but this can be changed by using `discount-on-abort`. If `discount-on-abort` is true, then `barrier-wait` decrements the arrivers count on aborting, so the call has no overall effect on the arrivers count.

If `barrier-wait` would have waited but `pass-through` is true, it returns the symbol `:passed-through` instead of waiting. Hence a call to `barrier-wait` with a true value of `pass-through` has the effect of incrementing the arriver count, and unblocking other waiters if needed, but never itself waiting.

Unblocking a barrier: when the number of arrivers at `barrier` equals its count minus 1, `barrier-wait` "unblocks the barrier". This involves the following steps:

1. If `callback` is non-nil, it is called with `barrier` while holding an internal lock in the barrier. See the comment in `make-barrier`. If `callback` aborts, nothing will have been changed in `barrier` (including no change to the number of arrivers).
2. `barrier` is marked as unblocked for the currently waiting threads.
3. The number of arrivers in `barrier` is reset to 0. Unless the next step disables `barrier`, this means that any subsequent call to `barrier-wait` will wait, as if `barrier` had just been created.
4. If `disable-on-unblock` is true, `barrier-wait` then disables `barrier`. Until it is re-enabled, any other call to `barrier-wait` will return immediately.
5. All the threads waiting on `barrier` are woken.
6. The symbol `:unblocked` is returned.

The possible values of `result` occur in these circumstances:
The current process waited and some other process unblocked \textit{barrier}.

\texttt{:unblocked} The current process unblocked \textit{barrier}.

\texttt{:timeout} The wait timed out.

\texttt{:passed-through} The current process did not wait because \textit{pass-through} is true.

\texttt{nil} \textit{barrier} is disabled.

\textbf{See also}\n\begin{itemize}
  \item \texttt{barrier}
  \item \texttt{barrier-arriver-count}
  \item \texttt{barrier-block-and-wait}
  \item \texttt{barrier-change-count}
  \item \texttt{barrier-count}
  \item \texttt{barrier-disable}
  \item \texttt{barrier-enable}
  \item \texttt{barrier-name}
  \item \texttt{barrier-pass-through}
  \item \texttt{barrier-unblock}
  \item \texttt{make-barrier}
  \item “Synchronization barriers” on page 287
\end{itemize}

\textbf{change-process-priority} \hfill \textit{Function}

\textbf{Summary} Changes the priority of a process.

\textbf{Package} \texttt{mp}

\textbf{Signature} \texttt{change-process-priority process new-priority => new-priority}

\textbf{Arguments} \begin{itemize}
  \item \texttt{process} A process.
  \item \texttt{new-priority} A fixnum.
\end{itemize}

\textbf{Description} Changes the priority of \texttt{process} to be \texttt{new-priority}. 
condition-variable

**System Class**

**Summary**
A class of objects for synchronizing processes.

**Package**
mp

**Superclasses**
t

**Description**
Instances of the system class *condition-variable* are used for synchronizing processes. They are made by *make-condition-variable*.

**See also**
make-condition-variable
condition-variable-wait
condition-variable-signal
“Condition variables” on page 285

condition-variable-broadcast

**Function**

**Summary**
Wakes all threads currently waiting on a given *condition-variable*.

**Package**
mp

**Signature**
condition-variable-broadcast condvar => signaledp

**Arguments**
condvar A *condition-variable*.

**Values**
signaledp A generalized boolean.

**Description**
The function *condition-variable-broadcast* wakes all threads currently waiting on the *condition-variable condvar*. In most uses of condition variables, the caller should
be holding the lock that the waiter used when calling condition-variable-wait for condvar, but this is not required. When using the lock, you may prefer to use lock-and-condition-variable-broadcast.

The return value signaledp is non-nil if some processes were signaled, or nil if there were no processes waiting.

See also

condition-variable-wait
make-condition-variable
lock-and-condition-variable-broadcast
lock-and-condition-variable-wait
simple-lock-and-condition-variable-wait
lock-and-condition-variable-signal
condition-variable-signal
“Condition variables” on page 285

**condition-variable-signal**

*Function*

*Summary*  Wakes one thread waiting on a given condition-variable.

*Package*  mp

*Signature*  condition-variable-signal condvar => signaledp

*Arguments*  condvar A condition-variable

*Values*  signaledp A generalized boolean

*Description*  The function condition-variable-signal wakes exactly one thread waiting on the condition-variable condvar. In most uses of condition variables, the caller should be holding the lock that the waiter used when calling condition-variable-wait for condvar, but this is not required. When using the lock, you may prefer to use lock-and-condition-variable-signal.
The return value \texttt{signaledp} is non-nil if a process was signaled, or \texttt{nil} if there were no processes waiting.

See also

condition-variable-wait
make-condition-variable
lock-and-condition-variable-signal
lock-and-condition-variable-wait
simple-lock-and-condition-variable-wait
lock-and-condition-variable-broadcast
condition-variable-broadcast
“Condition variables” on page 285

\section*{condition-variable-wait}

\textit{Function}

\begin{description}
\item[Summary] Waits for a given \texttt{condition-variable} to be signaled.
\item[Package] \texttt{mp}
\item[Signature] \texttt{condition-variable-wait condvar lock \&key timeout wait-reason => wakep}
\item[Arguments] \begin{itemize}
\item \texttt{condvar} A \texttt{condition-variable}.
\item \texttt{lock} A \texttt{lock}.
\item \texttt{timeout} A non-negative \texttt{real} or \texttt{nil}.
\item \texttt{wait-reason} A string.
\end{itemize}
\item[Values] \texttt{wakep} A generalized boolean.
\item[Description] The function \texttt{condition-variable-wait} waits at most \texttt{timeout} seconds for the \texttt{condition-variable condvar} to be signaled. The \texttt{lock} \texttt{lock} is released while waiting and claimed again before returning. The caller must be holding the \texttt{lock} \texttt{lock} before calling this function.
\end{description}
The return value \textit{wakeup} is non-nil if the signal was received or \textit{nil} if there was a timeout. If \textit{timeout} is \textit{nil}, \textit{condition-variable-wait} waits indefinitely.

If \textit{wait-reason} is non-nil, it is used as the \textit{wait-reason} while waiting for the signal.

It is recommended that you use \textit{lock-and-condition-variable-wait} or \textit{simple-lock-and-condition-variable-wait} instead of \textit{condition-variable-wait}. The locking functions make it easier to avoid mistakes, and can be more efficient.

\textbf{Notes} \hspace{1cm} \textit{timeout} controls how long to wait for the signal: before returning, the function waits to claim \textit{lock}, possibly indefinitely.

\textbf{See also} \hspace{1cm} \textit{condition-variable-wait-count}  
\textit{make-condition-variable}  
\textit{lock-and-condition-variable-wait}  
\textit{simple-lock-and-condition-variable-wait}  
\textit{lock-and-condition-variable-signal}  
\textit{lock-and-condition-variable-broadcast}  
\textit{condition-variable-signal}  
\textit{condition-variable-broadcast}  
“Condition variables” on page 285

\textbf{condition-variable-wait-count} \hspace{1cm} \textit{Function}

\textbf{Summary} \hspace{1cm} Returns the current number of threads that are still waiting for a \textit{condition-variable}.

\textbf{Package} \hspace{1cm} \textit{mp}

\textbf{Signature} \hspace{1cm} \textit{condition-variable-wait-count \hspace{2mm} condvar} \hspace{1mm}=> \hspace{1mm} \textit{wait-count}

\textbf{Arguments} \hspace{1cm} \textit{condvar} \hspace{1cm} A \textit{condition-variable}
Values

| wait-count | A non-negative integer |

Description

The function `condition-variable-wait-count` returns the current number of threads that are still waiting for `condvar`. Note that for a `condition-variable` that is actually in use, this number can change at any time.

See also

`condition-variable-wait`
“Condition variables” on page 285

---

*current-process*  

Variable

Summary

Contains the object that is the current process.

Package

`mp`

Description

This special variable contains the object that is the current process.

See also

`get-current-process`

---

current-process-block-interrupts

Function

Summary

Blocks interrupts in the current process.

Package

`mp`

Signature

`current-process-block-interrupts => t`

Description

The function `current-process-block-interrupts` blocks interrupts in the current process.

It signals an error if called outside the dynamic scope of `allowing-block-interrupts` or `with-interrupts-blocked`.
Blocking interrupts prevents any interruption of the current process, including process-interrupt, process-kill, process-reset, process-break and process-stop. These interrupts are all queued and processed once interrupts become unblocked.

Blocking interrupts also blocks interrupts due to POSIX signals. Such interrupts are processed either by another Lisp thread, or once interrupts become unblocked.

The effect of current-process-block-interrupts stays in force until the next call to either current-process-unblock-interrupts or current-process-block-interrupts, or an exit out of the scope of a surrounding allowing-block-interrupts or with-interrupts-blocked. Inside this range bodies of allowing-block-interrupts and with-interrupts-blocked have their own state, but they restore it on exit.

See also
allowing-block-interrupts
current-process-unblock-interrupts
process-break
process-interrupt
process-kill
process-reset
process-stop
with-interrupts-blocked

current-process-in-cleanup-p

Summary
The predicate for whether the current process is cleaning up after being killed.

Package
mp

Signature
current-process-in-cleanup-p => result
### current-process-in-cleanup-p

<table>
<thead>
<tr>
<th>Values</th>
<th>result</th>
<th>A boolean.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Description</th>
<th>The function <code>current-process-in-cleanup-p</code> returns true after the current process is killed. In particular, it returns true while the cleanups that were set by <code>ensure-process-cleanup</code> execute.</th>
</tr>
</thead>
</table>

| See also | `ensure-process-cleanup` |

### current-process-kill

<table>
<thead>
<tr>
<th>Summary</th>
<th>Kill the current process.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Package</th>
<th><code>mp</code></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Signature</th>
<th><code>current-process-kill</code></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
<th>None.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Values</th>
<th>None.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Description</th>
<th>The function <code>current-process-kill</code> kills the current process. <code>current-process-kill</code> signals an error if it is called when interrupts are blocked, unless it is inside the scope of <code>with-other-threads-disabled</code>, in which case the process is marked as &quot;dying&quot;, and actually dies on exit from <code>with-other-threads-disabled</code>. Normally, <code>current-process-kill</code> throws out and does not return. It does execute all surrounding <code>unwind-protect</code> forms. If <code>current-process-kill</code> is called while the process is already doing cleanups, it just returns.</th>
</tr>
</thead>
</table>
Notes

If you have a process that is broken and repeatedly goes into the debugger and you are not interested in debugging it, then calling \texttt{current-process-kill} is the best way of getting rid of it. This is especially useful on non-Cocoa platforms (GTK+ and Windows) when you get an interface that is badly broken.

See also \texttt{with-other-threads-disabled}

\section*{current-process-pause}

\textit{Function}

\begin{description}
\item[Summary] Sleeps for a specified time, but can be woken up.
\item[Package] \texttt{mp}
\item[Signature] \texttt{current-process-pause time \&optional function \&rest args => result}
\item[Arguments]
\begin{itemize}
\item \textit{time} A positive number.
\item \textit{function} A function designator.
\item \textit{args} Arguments passed to \textit{function}.
\end{itemize}
\item[Values] The keyword \texttt{poked}, or \texttt{nil}.
\item[Description] The function \texttt{current-process-pause} sleeps for \textit{time} seconds, but wakes up if another process did something to wake up the current process (normally this is \texttt{process-poke}, but it can also be \texttt{process-interrupt}, \texttt{process-stop}, \texttt{process-unstop} or \texttt{process-kill}).

\texttt{current-process-pause} is quite similar to \texttt{cl:sleep}, but it returns if anything causes the process to wake up, even if the time did not pass.

If \textit{function} is passed just before going to sleep, \texttt{current-process-pause} applies \textit{function} to \textit{args}, and if this returns a true value \texttt{current-process-pause} returns it immediately.
\end{description}
function and args are not used otherwise. If another process calls process-poke on the current process after setting something that causes function to return true, it guarantees that current-process-pause will return immediately without sleeping.

If another process woke up the current process, current-process-pause returns the keyword :poked. If it slept the full time, it returns nil.

Notes

1. In contrast to process-wait, the function argument to current-process-pause is applied only once, and within the dynamic scope of current-process-pause. It therefore does not have any of the restrictions that the wait-function of process-wait has.

2. The purpose of function is to guard against the possibility that another process pokes the current process while it is in the process of going to sleep.

3. There is no way to distinguish between the function returning :poked and the process being poked in some way.

4. The pausing does not happen reliably, and it can return :poked in a situation when it seems unexpected. For example, if the current process does:

```
(mailbox-read *mailbox*)
...
(current-process-pause)
```

the call to current-process-pause may return poked, because a process that sent an event to the mailbox tried to poke the current process, and by the time this poke happened the current process is already inside current-process-pause. The only guarantees are that current-process-pause does not wait when a poke occurred, and that it returns nil only when it paused the full time.
Example

Supposed you want to have a process that each minute does some cleanup, but may also be told by other processes to go and do the cleanup. The process be doing:

(loop
  (mp:current-process-pause 60 'check-for-need-cleanup)
  (do-cleanup))

Another process which wants to provoke a cleanup will do:

(setup-cleanup-flag)

(mp:process-poke *cleanup-process*)

Note that check-for-need-cleanup is passed to current-process-pause, because another process may call process-poke after current-process-pause was called but before it went to sleep. If check-for-need-cleanup was not passed, current-process-pause would unnecessarily sleep the whole 60 seconds in this case. The same thing could be implemented by process-wait-with-timeout, but the implementation above does not require a wait function that can run in another dynamic scope repeatedly at arbitrary times, and it uses much less system resources. It is also easier to debug.

See also  process-poke

**current-process-send**

*Function*

**Summary**

Sends an object as an event to the current process.

**Package**

mp

**Signature**

current-process-send object

**Arguments**

object A Lisp object.
The function `current-process-send` sends `object` as an event to the current process. This is useful when you want to execute code as an event rather than in the current context. A typical example is when a CAPI callback needs to do something in the current process which is not appropriate to invoke inside the callback.

For the object to actually be processed as an event, the current process must process events sometime after `current-process-send` is called. In the “standard” situation, for example in a process started by CAPI, the object will be processed as an event by calling `general-handle-event`.

See also `process-send`
`general-handle-event`
“Communication between processes and synchronization” on page 283

### `current-process-set-terminate-method` Function

**Summary**
Sets the Terminate Method of the current process.

**Package**
`mp`

**Signature**
`current-process-set-terminate-method &key local-terminator remote-terminator terminate-by-send`

**Arguments**
- `local-terminator` A function designator for a function of no arguments.
- `remote-terminator` A function designator for a function of one argument.
- `terminate-by-send` A generalized boolean.

**Description**
The function `current-process-set-terminate-method` sets the Terminate Method of the current process. See `pro-`
cess-run-function for the meaning of the keyword arguments.

The default value of terminate-by-send is t. Therefore calling current-process-set-terminate-method without arguments sets the Terminate Method to terminate-by-send. Calling current-process-set-terminate-method with terminate-by-send nil makes the process not have a Terminate Method.

See also
process-run-function
process-terminate
current-process-kill

current-process-unblock-interrupts

Function

Summary
Unblocks interrupts in the current process.

Package
mp

Signature
current-process-unblock-interrupts => t

Description
The function current-process-unblock-interrupts unblocks interrupts in the current process.

It signals an error if called outside the dynamic scope of allowing-block-interrupts or with-interrupts-blocked.

The effect of current-process-unblock-interrupts stays in force until the next call to either current-process-unblock-interrupts or current-process-block-interrupts, or an exit out of the scope of a surrounding allowing-block-interrupts or with-interrupts-blocked. Inside this range bodies of allowing-block-interrupts and with-interrupts-blocked have their own state, but they restore it on exit.
See also
allowing-block-interrupts
current-process-block-interrupts
with-interrupts-blocked

**debug-other-process**

*Function*

**Summary**
Examine the stack of a process other than the current process.

**Package**
mp

**Signature**
degug-other-process process

**Arguments**
process A process or a string.

**Description**
The function `debug-other-process` causes the debugger to be entered to examine the stack of another process `process`. The debugger itself continues to run in the current process, and the execution of the other process `process` is not affected. That means that all debugger commands that try to affect execution (for example :a, :c, :res, :ret, :trap) do not work as in the normal debugger. :a is changed instead to exit the debugger.

**Note:** if the other process is still active, the stack will change "under the feet" of the debugger, with unpredictable results. Thus `debug-other-process` is useful only for debugging purposes, or when you already stopped the other process.

The usual way to enter a debugger on another thread is to use `process-break`. However, that would fail if the other process hangs for some reason. In this situation, you can use `debug-other-process` to try to find out why it hangs.

If `process` is a string, the process is found as if by `find-process-from-name`. The list of process names can be found via `ps`. 
See also `find-process-from-name` `process-break` `ps`

*default-process-priority*  

**Variable**

**Summary**

The default priority for processes.

**Package**

`mp`

**Description**

The variable `*default-process-priority*` contains the default priority for processes.

See also `process-run-function`

**ensure-process-cleanup**  

**Function**

**Summary**

Run forms when a given process terminates.

**Package**

`mp`

**Signature**

```
ensure-process-cleanup cleanup-form &key priority force  
process =>
```

**Signature**

```
ensure-process-cleanup cleanup-form &optional process =>
```

**Arguments**

`cleanup-form`  
Form to run when `process` terminates.

`priority`  
An integer in the inclusive range `[-1000000, 1000000]`.

`force`  
A boolean.

`process`  
A `mp:process` object.

**Values**

None.
Description

The function `ensure-process-cleanup` ensures that the `cleanup-form` is present for the process `process`. When `process` terminates, its cleanup forms are run. Cleanup forms can be functions of one argument (the `process`), or lists, in which case the `cl:car` is applied to the `process` and the `cl:cdr` of the list.

`process` is the process to watch for termination. By default, this is the value returned by `get-current-process`.

`priority` determines the execution order of the forms. Higher `priority` means later execution. The system uses values between 700000 and 900000 for cleanups that need to be last, and 0 for other cleanups. The default value of `priority` is 0.

`force` determines what to do if the same cleanup is already registered but with a different `priority`. When adding cleanup forms, `ensure-process-cleanup` uses `cl:equal` to ensure that the form is only added once. If a cleanup already exists with the same priority, `ensure-process-cleanup` just returns `nil`, otherwise it acts according to `force`: if `force` is `nil` it invokes an error, but if `force` is `t` then `ensure-process-cleanup` removes the old entry before adding the new entry. The default value of `force` is `nil`.

When `ensure-processes-cleanup` is called on a foreign thread, that is a thread that was not created by LispWorks, the cleanups are executed after the outermost foreign-callable returns and before return to the foreign code that called it (that is when no Lisp frames remain on the stack).

Compatibility note

Before LispWorks 7.1, the cleanups where never executed when `ensure-processes-cleanup` was called in a foreign thread.

Notes

1. You can test for whether the current process is executing its cleanups with `current-process-in-cleanup-p`.

2. For compatibility with LispWorks 6.1 and earlier versions, `ensure-process-cleanup` can also be called like this:
(ensure-process-cleanup cleanup-form process)

Such calls are still allowed, for backwards compatibility, however please update your programs to call it like this:

(ensure-process-cleanup cleanup-form
 :priority priority
 :force force
 :process process)

Example
A process calls add-process-dependent each time a dependent object is added to a process. When the process terminates, inform-dependent-of-dead-process is called on all dependent objects.

(defun add-process-dependent (dependent)
  (mp:ensure-process-cleanup
   `(delete-process-dependent ,dependent)))

(defun delete-process-dependent (process dependent)
  (inform-dependent-of-dead-process dependent process))

See also
current-process-in-cleanup-p
process-terminate

find-process-from-name

Function

Summary
Finds a process from its name.

Package
mp

Signature
find-process-from-name process-name => result

Arguments
process-name A string.

Values
result A mp:process, or nil.

Description
The function find-process-from-name returns the process with the name process-name.

find-process-from-name
If there is no such process, the function returns `nil`.

**Example**

```
CL-USER 16 > (mp:find-process-from-name "Listener 1")
#<MP:PROCESS Name "Listener 1" Priority 600000 State "Running">
```

**See also**

`get-process`

### funcall-async

### funcall-async-list

**Summary**

Funcall a function asynchronously.

**Package**

`mp`

**Signature**

```
funcall-async func &rest args
funcall-async-list func-and-args
```

**Arguments**

- `func` A function designator.
- `args` Arguments.
- `func-and-args` A cons.

**Description**

The functions `funcall-async` and `funcall-async-list` call the function `func` with the supplied arguments, that is what `cl:funcall` would do, but asynchronously.

`func-and-args` must be a cons of a function designator and a proper list of arguments.

Multiprocessing must have already started.

These functions do not return a useful value.

**Notes**

1. These functions are effectively lightweight versions of `process-run-function`.
2. On most architectures they are implemented using worker processes, which are named "Background Execute n".

3. The maximum number background processes is limited by default to 5 and this is adequate in most cases. However, if you use `funcall-async` and/or `funcall-async-list` often, you may want to increase the limit, by using `set-funcall-async-limit`.

4. The dynamic context of the call to `func` is undefined, and must not be relied upon.

5. The current process should not be accessed inside `func`, except when you want another process to poke the process that runs `func` (this is sometimes useful if `func` calls a wait function). In this case you can call `get-current-process` inside the dynamic scope of `func` to get the process that the other process should poke.

6. `funcall-async` and `funcall-async-list` are intended for functions that finish quickly. If `func` takes a long time, it prevents the background process from executing other async calls, and if all of the background processes become occupied by long-executing functions it will cause other async calls to be delayed until one of the background processes finishes. Thus if you have a long-executing function that you want to run asynchronously, it is better to use `process-run-function` instead, or use your own pool of worker processes.

See also

- `process-run-function`
- `set-funcall-async-limit`

**general-handle-event**  
*Generic function*

**Summary**  
"handles" an event, depending on the type of the event object.
Package  mp

Signature  general-handle-event  event-object

Arguments  event-object  A Lisp object.

Description  The generic function general-handle-event "handles" the event-object. What this actually means depends on the type of the object. There are system defined methods for these classes:

list  Apply the car to the cdr.
function  Call it.
symbol  If fbound call it, otherwise do nothing.
t  Do nothing.

You can add methods for your own classes.

general-handle-event is used by all functions that process events, for example wait-processing-events and process-all-events, as well as by internal waiting functions.

See also  process-all-events
process-send
"Communication between processes and synchronization" on page 283

get-current-process  Function

Summary  Returns the current Lisp process.

Package  mp

Signature  get-current-process => result

Values  result  A mp:process, or nil.
The function `get-current-process` returns the actual process in which it is called. In this respect it differs from `*current-process*`, which can be bound to another process. In particular, when a process A calls the `wait-function` of process B, in the `wait-function` `get-current-process` returns the process A, but `*current-process*` is bound to process B.

`result` is `nil` if multiprocessing is off.

See also `*current-process*`
A symbol
First search for a process using the symbol name as a string, and (if that fails) then search using the symbol as a function.

A fixnum
Find a process for which process-designator is its unique id. The unique id of the current process can be found by (sys:current-thread-unique-id).

result is nil if multiprocessing is off.

See also find-process-from-name

get-process-private-property

Function

Summary
Gets the value of a process private property.

Package
mp

Signature
get-process-private-property indicator process &optional default => result

Arguments
indicator A Lisp object.
process A process.
default A Lisp object.

Values
result A property value, or default.

Description
The function get-process-private-property gets the value associated with indicator in the private properties of the process process. If there is no such property, the value default is returned.

get-process-private-property can be used to read the values of private properties from another process.

The default value of default is nil.
Function

initialize-multiprocessing

Summary
Initializes multiprocessing before use.

Package
mp

Signature
initialize-multiprocessing &rest main-process-args => nil

Arguments
main-process-args
A set of arguments for process-run-function.

Values
Returns nil.

Description
The function initialize-multiprocessing initializes multiprocessing, and it does not return until multiprocessing is finished.

initialize-multiprocessing applies the function process-run-function to each of the entries in *initial-processes* to create the initial processes.

When called with main-process-args, it creates a mp:process object for the initial thread using the arguments in that list as if in the call

(apply 'mp:process-run-function main-process-args)

Supplying main-process-args is useful on Mac OS X if you want to run a pure Cocoa application, since the main thread needs to run the Cocoa event loop.
It is not necessary to call `initialize-multiprocessing` when the LispWorks IDE is running (that is, after `env:start-environment` has been called), as this automatically starts up multiprocessing.

**Notes**

On Microsoft Windows, Linux, x86/x64 Solaris, FreeBSD, AIX and Mac OS X (using the Cocoa image), the LispWorks IDE starts up by default.

**See also**

*initial-processes*

`process-run-function`

### *initial-processes* Variable

**Summary**

A list of the processes the system initializes on startup.

**Package**

mp

**Description**

The variable *initial-processes* specifies the processes which the system initializes on startup.

Each element of the *initial-processes* list is a set of arguments for `process-run-function`.

**Example**

To create a listener process as well as your own processes, evaluate this form before saving your image:

```
(push mp::*default-listener-process*
  mp:*initial-processes*)
```

**See also**

`process-run-function`
last-callback-on-thread  

**Function**

**Summary**
Informs LispWorks that there are probably not going to be more callbacks from foreign code on the current thread, allowing it to free some data.

**Package**
mp

**Signature**
last-callback-on-thread => result

**Values**
result  t or nil.

**Description**
The function last-callback-on-thread informs LispWorks that there are probably not going to be more callbacks from foreign code on the current thread (but does not guarantee this).

last-callback-on-thread must be used in the scope of a call into LispWorks by a foreign callable on a thread that was not created by LispWorks. It informs LispWorks that there are unlikely to be more callbacks into Lisp on the current thread. As a result, LispWorks can cleanup its side.

For each thread that was not created by Lisp and on which there was a call into Lisp, LispWorks keeps data on the Lisp side which it uses to make the entry faster. If the thread goes away, this data is not needed and so LispWorks can free it.

If another callback occurs on the same thread after a callback that called last-callback-on-thread, LispWorks will have to recreate its side, which takes a little more time, but otherwise it works in the same way. Thus it is possible to call last-callback-on-thread even when it is not guaranteed that there will not be further callbacks on the same thread.

Calling last-callback-on-thread on a thread that was created by LispWorks has no effect.
last-callback-on-thread returns t when called on a thread that was not created by LispWorks, otherwise it returns nil.

list-all-processes

Summary
Lists all the Lisp processes currently in the system.

Package
mp

Signature
list-all-processes => process-list

Arguments
None.

Values
process-list
A list of all the currently active Lisp processes.

Description
Returns a list of all the active Lisp processes in LispWorks.

Example
CL-USER 71 > (pprint (mp:list-all-processes))

(#<MP:PROCESS Name "Editor 1" Priority 70000000 State "Waiting for events">
  #<MP:PROCESS Name "Listener 1" Priority 70000000 State "Running">
  #<MP:PROCESS Name "LispWorks 5.1.0" Priority 70000000 State "Waiting for events">
  #<MP:PROCESS Name "default listener process" Priority 60000000 State "Waiting for terminal input.">
  #<MP:PROCESS Name "CAPI Execution Listener 1" Priority 60000000 State "Running">
  #<MP:PROCESS Name "Background execute 2" Priority 50000000 State "Waiting for job to execute">
  #<MP:PROCESS Name "Background execute 1" Priority 50000000 State "Waiting for job to execute">
  #<MP:PROCESS Name "Editor DDE server" Priority 0 State "Waiting for an event">
  #<MP:PROCESS Name "The idle process" Priority -536870912 State "Running (preempted)">)
**lock**  
*System Class*

**Summary**  
A class of objects for preventing synchronous access.

**Package**  
mp

**Superclasses**  
t

**Description**  
Instances of the system class **lock** are used to prevent synchronous access to the some object(s) by more than one process at a time. They are made by **make-lock**.

**See also**  
make-lock  
with-lock  
process-lock  
process-unlock  
“Locks” on page 277

---

**lock-and-condition-variable-broadcast**  
*Function*

**Summary**  
Locks, applies a setup function, calls **condition-variable-broadcast** and unlocks.

**Package**  
mp

**Signature**  
lock-and-condition-variable-broadcast lock condvar lock-timeout setup-function &rest args => signaledp

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td>A lock.</td>
</tr>
<tr>
<td>condvar</td>
<td>A <strong>condition-variable</strong>.</td>
</tr>
<tr>
<td>lock-timeout</td>
<td>A non-negative <strong>real</strong> or <strong>nil</strong>.</td>
</tr>
<tr>
<td>setup-function</td>
<td>A function designator.</td>
</tr>
<tr>
<td>args</td>
<td>Arguments to <strong>setup-function</strong>.</td>
</tr>
</tbody>
</table>

**Values**  
signaledp  
A generalized boolean.
Description

The function `lock-and-condition-variable-broadcast` locks the `lock` `lock`, applies the function `setup-function` to `args`, calls `condition-variable-broadcast` and unlocks `lock`.

`lock-and-condition-variable-broadcast` makes it easier to avoid mistakes when using a `condition-variable`.

`lock-and-condition-variable-broadcast` performs the equivalent of:

```lisp
(mp:with-lock (lock nil lock-timeout)
   (apply setup-function args)
   (mp:condition-variable-broadcast condvar))
```

It returns the result of the call to `condition-variable-broadcast`.

See `condition-variable-broadcast` and `with-lock` for more details.

Notes

`setup-function` is called with the `lock` held, so it should do the minimum amount of work and avoid locking other locks.

See also

`lock-and-condition-variable-wait`
`simple-lock-and-condition-variable-wait`
`lock-and-condition-variable-signal`
`condition-variable-wait`
`condition-variable-signal`
`condition-variable-broadcast`
`processes-count`
`with-lock`

“Condition variables” on page 285
“Locks” on page 277

`lock-and-condition-variable-signal` Function

Summary

Locks, applies a setup function, calls `condition-variable-signal` and unlocks.
Arguments

- **lock**: A lock.
- **condvar**: A condition-variable.
- **lock-timeout**: A non-negative real or nil.
- **setup-function**: A function designator.
- **args**: Arguments to setup-function.

Values

- **signaledp**: A generalized boolean.

Description

The function `lock-and-condition-variable-signal` locks the lock `lock`, applies `setup-function` to `args`, calls `condition-variable-signal` and unlocks `lock`. `lock-and-condition-variable-signal` makes it easier to avoid mistakes when using a condition-variable.

`lock-and-condition-variable-signal` performs the equivalent of:

```lisp
(mp:with-lock (lock nil lock-timeout)
  (apply setup-function args)
  (mp:condition-variable-signal condvar))
```

It returns the result of the call to `condition-variable-signal`.

See `condition-variable-signal` and `with-lock` for more details.

Notes

`setup-function` is called with `lock` held, so it should do the minimum amount of work and avoid locking other locks. Normally `setup-function` should only set the cell that the process(es) that wait(s) on the condition-variable `condvar` check with the predicate in `lock-and-condition-variable-wait`.
See also

lock-and-condition-variable-wait
simple-lock-and-condition-variable-wait
lock-and-condition-variable-broadcast
condition-variable-wait
condition-variable-signal
condition-variable-broadcast
“Condition variables” on page 285
“Locks” on page 277

**lock-and-condition-variable-wait**

**Function**

**Summary**
Locks a lock and calls a predicate. If this returns `nil`, performs the equivalent of `condition-variable-wait`. Optionally calls a function on return.

**Package**
mp

**Signature**

```
lock-and-condition-variable-wait lock condvar predicate &key args return-function return-function-args lock-timeout lock-wait-reason condvar-timeout condvar-wait-reason => result
```

**Arguments**

- `lock` A lock.
- `condvar` A `condition-variable`.
- `predicate` A function designator.
- `args` Arguments to `predicate`.
- `return-function` A function designator or `nil`.
- `return-function-args` Arguments to `return-function`.
- `lock-timeout` A non-negative `real` or `nil`.
- `lock-wait-reason` A `string` or `nil`.
- `condvar-timeout` A non-negative `real` or `nil`.
- `condvar-wait-reason`
A string or nil.

Values

|        | result | See below. |

Description

The function `lock-and-condition-variable-wait` first locks the `lock lock` as in `with-lock`, using `lock-wait-reason` and `lock-timeout` for the `whostate` and `timeout` arguments of `with-lock`.

It then applies `predicate` to `args`. If this call returns `nil` it performs the equivalent of a call to `condition-variable-wait`, passing it `condvar`, `lock`, `condvar-timeout` and `condvar-wait-reason`.

If `return-function` is supplied, it is then applied to `return-function-args`, and the return value(s) are returned.

Before returning, `lock` is unlocked (in an unwinding form) as in `with-lock`.

`lock-and-condition-variable-wait` returns whatever `return-function` returns if it is supplied. If `return-function` is not supplied, `lock-and-condition-variable-wait` returns the result of `predicate` if this is not `nil`, otherwise it returns the result of the equivalent call to `condition-variable-wait`.

Notes

1. `predicate` and `return-function` are called with `lock` held, so they should do as little as needed, and avoid locking anything else.

2. `lock-and-condition-variable-wait` makes it much easier to avoid mistakes when using `condition-variables`.

3. When `return-function` is not supplied, `lock-and-condition-variable-wait` does not lock on return, which makes it much more efficient than the equivalent code using `with-lock` and `condition-variable-wait`.

4. When `return-function` is not needed, `simple-lock-and-condition-variable-wait` may be more convenient.
5. All the four signaling functions (\texttt{condition-variable-signal}, \texttt{condition-variable-broadcast}, \texttt{lock-and-condition-variable-signal}, \texttt{lock-and-condition-variable-broadcast}) can be used to wake a process waiting in \texttt{lock-and-condition-variable-wait}.

See also

\begin{itemize}
  \item \texttt{condition-variable-wait}
  \item \texttt{simple-lock-and-condition-variable-wait}
  \item \texttt{lock-and-condition-variable-signal}
  \item \texttt{lock-and-condition-variable-broadcast}
  \item \texttt{condition-variable-signal}
  \item \texttt{condition-variable-broadcast}
  \item “Condition variables” on page 285
  \item “Locks” on page 277
\end{itemize}

\textbf{lock-locked-p}

\textit{Function}

\textbf{Summary}

The predicate for whether a lock is locked.

\textbf{Package}

\texttt{mp}

\textbf{Signature}

\texttt{lock-locked-p} \texttt{lock => result}

\textbf{Arguments}

\texttt{lock} \text{A lock.}

\textbf{Values}

\texttt{result} \text{A boolean.}

\textbf{Description}

The function \texttt{lock-locked-p} is the predicate for whether \texttt{lock} is locked. Since that can change at any time, the result is reliable only if you know that the state is not going to change.

If \texttt{lock} is a "sharing" lock, this checks whether it is locked exclusively.

See also

\begin{itemize}
  \item \texttt{lock}
  \item \texttt{make-lock}
  \item “Locks” on page 277
\end{itemize}
**lock-owned-by-current-process-p**

**Function**

**Summary**
Checks whether a lock is locked by the current thread.

**Package**
mp

**Signature**
lock-owned-by-current-process-p lock => result

**Arguments**
lock A lock.

**Values**
result A boolean.

**Description**
The function lock-owned-by-current-process-p checks whether the lock lock is locked by the current thread. If this returns nil, then lock is either unlocked or locked by another process.

If lock is a "sharing" lock, this also checks whether the current process has an exclusive lock on it. It ignores any shared lock.

**See also**
lock
make-lock
“Locks” on page 277

**lock-recursive-p**

**Function**

**Summary**
The predicate for whether a lock allows recursive locking.

**Package**
mp

**Signature**
lock-recursive-p lock => result

**Arguments**
lock A lock.

**Values**
result A boolean.
The function `lock-recursive-p` is the predicate for whether the lock allows recursive locking (that is, whether it can be repeatedly locked by the same process).

See the `make-lock` argument `recursivep`.

`lock-recursive-p` does not check whether lock is currently locked recursively. The function `lock-recursively-locked-p` does that.

See also
- `lock`
- `make-lock`
- “Locks” on page 277

**lock-recursively-locked-p**

Function

Summary
The predicate for whether a lock is recursively locked.

Package
mp

Signature
`lock-recursively-locked-p` `lock` `=>` `result`

Arguments
`lock` A lock.

Values
`result` A boolean.

Description
The function `lock-recursively-locked-p` is the predicate for whether `lock` is recursively locked. Since that can change at any time, the result is reliable only if you know that the state is not going to change. For the definition of recursive locking, see the `make-lock` argument `recursivep`.

If `lock` is a “sharing” lock, `lock-recursively-locked-p` checks whether is is locked exclusively.
See also

lock
make-lock
“Locks” on page 277

lock-name  

Function

Summary  Returns the name of a lock.

Package  mp

Signature  lock-name lock => name

Arguments  lock  A lock.

Values  name  A string.

Description  The function lock-name returns the name of lock, which was either passed as the name argument to make-lock or defaulted.

Example  

(let ((lock (mp:make-lock :name "my lock")))
  (mp:lock-name lock))

=> "my lock"

See also  

lock
make-lock
with-lock
process-lock
process-unlock
lock-owner
“Locks” on page 277

lock-owner  

Function

Summary  Returns the owner of a lock.
The MP Package

Package  mp

Signature  lock-owner lock => result

Arguments  lock        A lock.

Values     result      A process, t or :unknown.

Description The function lock-owner returns the process that currently
owns lock, or nil.

If lock is a "sharing" lock then lock-owner checks whether it
is locked exclusively (see lock-owned-by-current-process-p).

If lock is locked then result is normally the process that locked
it. If lock was locked while multiprocessing was not running
then result is t. Also, if lock was locked by an unknown pro-
cess (for example, the process is killed while holding lock)
then result is :unknown.

result is nil if lock is not locked.

Example

CL-USER 1 > (let ((lock (mp:make-lock :name
                       "my lock")))
              (mp:lock-owner lock))
NIL

CL-USER 2 > (let ((lock (mp:make-lock :name
                       "my lock")))
              (mp:with-lock (lock)
                  (mp:lock-owner lock)))
#<MP:PROCESS Name "CAPI Execution Listener 1" Priority
  0 State "Running">

See also  lock
          lock-owned-by-current-process-p
          make-lock
          with-lock
          process-lock
          process-unlock
**mailbox**

*System Class*

**Summary**
The class of objects representing mailboxes.

**Package**
mailbox

**Superclasses**
t

**Description**
Instances of the system class mailbox are used to communicate between processes. The communication is done by "sending" objects (any Lisp object) to a mailbox, and "reading" these objects from the mailbox. The objects will be read in the order in which they were sent. Sending is done by mailbox-send or mailbox-send-limited. Reading is done by mailbox-wait-for-event, mailbox-wait or mailbox-read. All mailbox access functions are thread-safe. You create a mailbox by using make-mailbox. You can also obtain the mailbox of a process by process-mailbox.

**See also**
mailbox-send
mailbox-send-limited
mailbox-wait-for-event
mailbox-wait
mailbox-read
make-mailbox
“Communication between processes and synchronization” on page 283

**mailbox-count**

*Function*

**Summary**
Returns the number of objects currently in a mailbox.
The MP Package

Package mp

Signature mailbox-count mailbox => count

Arguments mailbox A mailbox.

Values count A non-negative integer.

Description The function mailbox-count returns the number of objects currently in the mailbox mailbox. mailbox should be an object of type mailbox. A mailbox is empty if its count is 0.

See also mailbox-empty-p
mailbox-not-empty-p
make-mailbox
“Communication between processes and synchronization” on page 283

mailbox-empty-p Function

Summary Tests whether a mailbox is empty.

Package mp

Signature mailbox-empty-p mailbox => bool

Arguments mailbox A mailbox

Values bool A boolean

Description The function mailbox-empty-p returns t if the given mailbox is empty and nil otherwise.
mailbox-full-p

Function

Summary
Tests whether a mailbox is full.

Package
mp

Signature
mailbox-full-p mailbox => bool

Arguments
mailbox A mailbox

Values
bool A boolean

Description
The function mailbox-full-p returns true if mailbox is empty and false otherwise.

Notes
Because of multiprocessing, it cannot be guaranteed that a subsequent call to mailbox-send will work without expansion even if mailbox-full-p returns false. mailbox-full-p is intended to be used as a wait-function (or inside a wait-function), and once it returns false you should use mailbox-send-limited to actually send and check what it returns.

See also
mailbox-send-limited
mailbox-size
mailbox-count
“Communication between processes and synchronization” on page 283
The function `mailbox-not-empty-p` returns `nil` if the given `mailbox` is empty and `t` otherwise.

See also `mailbox-count`, `mailbox-empty-p`, `mailbox-send`, `mailbox-peek`, `mailbox-read`, `make-mailbox`.

---

`mailbox-peek` Function

Summary

Returns the first object in a `mailbox`.

Package `mp`

Signature

`mailbox-peek mailbox => result, value-p`

Arguments `mailbox` A `mailbox`.

Values `result` Any object or `nil`.

`value-p` `t` or `nil`.
The function `mailbox-peek` returns the first object in the mailbox without removing it. If the mailbox is empty, `nil` is returned.

If the mailbox `mailbox` is not empty, the function `mailbox-peek` returns the first object in the mailbox without removing it. The second returned value `value-p` is `t`.

If `mailbox` is empty, both return values `result` and `value-p` are `nil`.

Notes

1. Since another process may modify the mailbox at any point, the result is not necessarily the next object that the next call to `mailbox-read` will read, unless no other process is reading from the mailbox.

2. `mailbox-peek` needs to lock the mailbox, which means it is significantly slower than `mailbox-not-empty-p`, and also may affect other processes. In most cases, `mailbox-not-empty-p` is sufficient and hence is preferable.

See also

- `mailbox-empty-p`
- `mailbox-not-empty-p`
- `mailbox-send`
- `mailbox-read`
- `make-mailbox`
- “Communication between processes and synchronization” on page 283

### mailbox-read

**Function**

**Summary**
Reads the next object in a `mailbox`.

**Package**
`mp`

**Signature**
`mailbox-read mailbox &optional wait-reason timeout => object, flag`
The MP Package

Arguments

- **mailbox**: A mailbox.
- **wait-reason**: A string or **nil**.
- **timeout**: A non-negative **real** or **nil**.

Values

- **object**: An object.
- **flag**: A boolean.

Description

The function **mailbox-read** returns the next object from the mailbox **mailbox**, or **nil**.

If **mailbox** is empty and **timeout** is **nil**, then **mailbox-read** blocks until an object is placed in **mailbox**. If **mailbox** is empty and **timeout** is a non-negative **real**, then **mailbox-read** blocks until an object is placed in **mailbox** or **timeout** seconds have passed. If the timeout occurs, then **mailbox-read** returns **nil** as the first value and also **flag** is **nil**. If an object is actually read from the mailbox, then **flag** is **t**.

The **wait-reason** argument defaults to "Waiting for message in #<Mailbox...>" and will be the value returned by **process-whostate** while **mailbox-read** is blocking.

The default value of **timeout** is **nil**.

See also

- **mailbox-empty-p**
- **mailbox-peek**
- **mailbox-send**
- **mailbox-wait-for-event**
- **make-mailbox**
  “Communication between processes and synchronization” on page 283

**mailbox-reader-process**

Function

Summary

Returns the reader process of a **mailbox**.
The function `mailbox-reader-process` returns the reader process of `mailbox`.

**mailbox-send**

*Function*

**Summary**

Adds an object to a `mailbox`.

**Package**

`mp`

**Signature**

`mailbox-send mailbox object =>`

**Arguments**

`mailbox` A `mailbox`.

`object` An object.

**Description**

The function `mailbox-send` sends `object` to `mailbox`. The object is queued in the mailbox for retrieval by the reader. When objects are read from from the mailbox (using `mailbox-read` or higher level functions such as `mailbox-wait-for-event`), they are read in the order in which they were added to the mailbox.

**Notes**

If `mailbox` is full, `mailbox-send` expands it. In situations where the mailbox can grow too much, you can use `mailbox-send-limited` instead.

**See also**

`mailbox-empty-p`

`mailbox-peek`
mailbox-read
mailbox-send-limited
mailbox-count
mailbox-size
process-send
make-mailbox

“Communication between processes and synchronization” on page 283

**mailbox-send-limited**

**Function**

**Summary**

Adds an object to a **mailbox** if it is not full

**Package**

mp

**Signature**

`mailbox-send-limited mailbox object limit &optional timeout wait-reason wait-function &rest args => success`

**Arguments**

- `mailbox` A mailbox.
- `object` An object.
- `limit` An integer.
- `timeout` A non-negative real or nil.
- `wait-reason` A string or nil.
- `wait-function, args` A function and its arguments.

**Values**

- `success` A boolean.

**Description**

The function **mailbox-send-limited** adds **object** to **mailbox** in the same way as **mailbox-send**, except in the case where **mailbox** is full.
If mailbox is full and has a size less than limit then mailbox is enlarged to have a size that is at most limit and object is added to mailbox.

Otherwise, if mailbox is full and has a size not less than limit then mailbox-send-limited waits until mailbox becomes not full before adding object. While waiting, mailbox-send-limited will return without adding object to mailbox if timeout is non-nil and timeout seconds has elapsed or if wait-function is non-nil and applying wait-function to args returns true. wait-reason is used as the wait-reason while waiting.

success is true if the object was added to the mailbox and false otherwise (timeout reached or wait-function returned true).

Notes

mailbox-send-limited only prevents the mailbox from expanding to more than limit: if is already bigger than the limit, and there is still a space in it, mailbox-send-limited add the object to the mailbox even if that means that the mailbox has more objects than limit. As long as all of the sending calls on a mailbox are limited, the mailbox may grow until it reaches the largest limit, and if it was made with the :size argument equal or larger than the largest limit, it will never grow. However, if mailbox-send is called then that may enlarge it.

process-send uses mailbox-send, so may expand the mailbox too. It also has a limit argument.

mailbox-send-limited waits like process-wait-with-timeout. As a result, there may be some latency between the time the mailbox becomes non-full and the waiting returns, because it depends on the scheduler. The wait-reason, timeout, wait-function and args are analogous to the arguments of process-wait-with-timeout. wait-function is applied within the wait-function that mailbox-send-limited uses, so has the same limitations as the wait-function of process-wait-with-timeout.
When `timeout` is 0 `mailbox-send-limited` never waits.

See also

- `mailbox-send`
- `mailbox-read`
- `process-send`
- `mailbox-count`
- `mailbox-size`
- `mailbox-full-p`

“Communication between processes and synchronization” on page 283

---

**mailbox-size**

**Function**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Returns the size of a <code>mailbox</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td><code>mp</code></td>
</tr>
<tr>
<td>Signature</td>
<td><code>mailbox-size mailbox =&gt; size</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>mailbox</code></td>
</tr>
<tr>
<td></td>
<td>A <code>mailbox</code>.</td>
</tr>
<tr>
<td>Values</td>
<td><code>size</code></td>
</tr>
<tr>
<td></td>
<td>An non-negative integer.</td>
</tr>
</tbody>
</table>

**Description**

The function `mailbox-size` returns the size of `mailbox`, which is the number of objects that can be added without it growing.

`mailbox-send` automatically expands the mailbox indefinitely when it is full, but `mailbox-send-limited` and `process-send` will expand it up to some specified `limit`.

See also

- `mailbox-send`
- `mailbox-send-limited`
- `process-send`
- `make-mailbox`

“Communication between processes and synchronization” on page 283
**Function**

**mailbox-wait**

**Summary**
Waits until there is an object in the mailbox

**Package**
mp

**Signature**
mailbox-wait mailbox &optional wait-reason timeout => result

**Arguments**

- **mailbox**
  A mailbox.

- **wait-reason**
  A string or nil.

- **timeout**
  A non-negative real or nil.

**Values**

- **result**
  A boolean.

**Description**
The function mailbox-wait waits until there is an object in the mailbox mailbox.

If mailbox is empty and timeout is nil, then mailbox-wait blocks until an object is placed in mailbox. If mailbox is empty and timeout is a non-negative real, then mailbox-wait blocks until an object is placed in mailbox or timeout seconds have passed. If there is no object after timeout seconds, then mailbox-wait returns nil. Once there is an object in mailbox, mailbox-wait returns t.

Note that mailbox-wait does not remove the object from the mailbox, in contrast to mailbox-read which does.

Note that if there are multiple processes reading from mailbox, another process may read the object from it, so the result of mailbox-wait is reliable only if you know that the current process is the only process that may read from the mailbox.

The wait-reason argument defaults to a string

"Waiting for message in #<Mailbox...>"

and will be the value returned by process-whostate while mailbox-wait is blocking.
The default value of *timeout* is `nil`.

*mailbox-wait* arranges for immediate notification when an object is placed in *mailbox* (unless other processes are waiting too, in which case one of the processes is notified immediately). It is therefore better than using *process-wait* with *mailbox-not-empty-p* because it does not rely on the scheduler to wake it up. It is also less expensive because it does not add work to the scheduler.

See also  
- *mailbox-not-empty-p*  
- *mailbox-empty-p*  
- *mailbox-peek*  
- *mailbox-send*  
- *mailbox-wait-for-event*  
- *mailbox-read*

“Communication between processes and synchronization” on page 283

### mailbox-wait-for-event

**Function**

**Summary**

Waits for an event in a "windowing friendly" way.

**Package**

`mp`

**Signature**

```lisp
mailbox-wait-for-event mailbox &key wait-reason wait-function process-other-messages-p no-hang-p stop-at-user-operation-p => result
```

**Arguments**

- `mailbox`  
  A mailbox.
- `wait-reason`  
  A string or `nil`.
- `wait-function`  
  A function designator.
- `process-other-messages-p`  
  A generalized boolean.
- `no-hang-p`  
  A generalized boolean.
*stop-at-user-operation-p*

A generalized boolean.

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>An event or nil.</td>
</tr>
</tbody>
</table>

The function `mailbox-wait-for-event` waits for an event in a mailbox in a "windowing friendly" way. It reads an event from the mailbox `mailbox`. If there is no event in the mailbox, it waits for an event (unless `no-hang-p` is true).

The value `result` is any object that was put in the mailbox, or `nil` if the mailbox is empty, possibly after waiting.

`mailbox-wait-for-event` is the appropriate way to wait for an event in a mailbox in an application with a graphical user interface, because it interacts correctly with the windowing system. Most importantly, on Microsoft Windows, when `process-other-messages-p` is true it processes Windows messages while it is waiting. The default value of `process-other-messages-p` is `t`.

If `wait-function` is non-nil, then is it called as a Process Wait function (see “Generic Process Wait functions” on page 281) with the mailbox `mailbox` as its argument while waiting for an event. If the call returns true before any events arrive, then `mailbox-wait-for-event` will return `nil`.

`wait-reason` is used as the wait reason if it needs to wait. The default value of `wait-reason` is "Waiting for an event".

`process-other-messages-p` controls processing of other messages. On Microsoft Windows this means Windows messages. On other platforms it has no effect.

`no-hang-p` controls whether `mailbox-wait-for-event` should really wait. If `no-hang-p` is true and there is no event, it returns immediately except on Microsoft Windows, where it may first process all Windows messages (depending on the value of `process-other-messages-p`). The default value of `no-hang-p` is `nil`. 
stop-at-user-operation-p on Microsoft Windows causes mailbox-wait-for-event to return if it received a user operation message (meaning keyboard or mouse input). It has no effect on other platforms. The default value of stop-at-user-operation-p is nil.

If mailbox-wait-for-event is called when not Lisp is not multiprocessing, it returns immediately. The return value is an event or nil.

See also mailbox-read
mailbox-send
make-mailbox
process-wait-for-event
"Communication between processes and synchronization" on page 283

*main-process* Variable

Summary The process associated with the main thread.

Package mp

Description This special variable contains the process associated with the main thread of the application. On Mac OS X with the Cocoa GUI, this is the thread that runs the Cocoa event loop. On other platforms, this variable is always nil.

make-barrier Function

Summary Returns a new barrier.

Package mp

Signature make-barrier count &key discount-on-abort discount-on-timeout callback disable-on-unblock name => barrier
Arguments

- **count**: A positive fixnum or t.
- **name**: A string.

Values

- **barrier**: A barrier.

Description

The function `make-barrier` returns a new barrier with count `count`.

`count` can be t, which is interpreted as `most-positive-fixnum`.

`barrier` has the name `name`, which is useful for debugging but is not used otherwise. If `name` is omitted, then a default name is generated that is unique among all such default names.

`discount-on-timeout` and `discount-on-abort` determine the default behavior when a thread times out or aborts while in the function `barrier-wait`. See the documentation for `barrier-wait`.

If `disable-on-unblock` is true, then `barrier-wait` will disable `barrier` by default when it unblocks it. See `disable-on-unblock` in the documentation for `barrier-wait`.

`callback` is called by `barrier-wait` just before it unblocks `barrier`. It is called with a single argument, `barrier`, while holding an internal lock in `barrier` that will prevent other operations on `barrier` from running. The callback is guaranteed to happen before `barrier-wait` allows any of the other threads to continue.

Notes

Because the callback is called inside a lock, you should ensure that it is relatively short to prevent congestion if another thread tries to access `barrier`. Allocating a few objects is reasonable. If there is a more expensive operation that has to be done by only one of the threads, it can be done by the thread that returned :unblocked from `barrier-wait`. The advantage of using the callback is that it is called before any of the waiting threads pass the barrier.
See also
barrier
barrier-arriver-count
barrier-block-and-wait
barrier-change-count
barrier-count
barrier-disable
barrier-enable
barrier-name
barrier-pass-through
barrier-unblock
barrier-wait
“Synchronization barriers” on page 287

make-condition-variable

Function

Summary
Makes a condition-variable.

Package
mp

Signature
make-condition-variable &key name => condvar

Arguments
name A string naming the condition-variable.

Values
condvar A condition-variable.

Description
The function make-condition-variable makes a condition-variable for use with condition-variable-wait, condition-variable-signal and condition-variable-broadcast.

name is used when printing the condition-variable, and is useful for debugging. If name is omitted, then a default name is generated that is unique among all such default names.

See also
condition-variable
condition-variable-wait
condition-variable-signal
make-lock  

**Summary**  
Makes a lock.

**Package**  
mp

**Signature**  
make-lock &key name important-p safep recursivep sharing => lock

**Arguments**
- name: A string.
- important-p: A generalized boolean.
- safep: A generalized boolean.
- recursivep: A generalized boolean.
- sharing: A generalized boolean.

**Values**
- lock: A lock.

**Description**  
The function make-lock creates a lock. See “Locks” on page 277 for a general description of locks.

name names lock and can be queried with lock-name. The default value of name is "Anon".

important-p controls whether lock is automatically freed when the holder process finishes. When important-p is true, the system notes that lock is important, and automatically frees it when the holder process finishes. important-p should be nil for locks which are managed completely by the application, as it is wasteful to record all locks in a global list if there is no need to free them automatically. This might be appropriate when two processes sharing a lock must both be running for the system to be consistent. If one process dies, then the other one kills itself. Thus the system does not need to worry about
freeing the lock because no process could be waiting on it forever after the first process dies. The default value of important-p is nil.

safep controls whether lock is safe. A safe lock gives an error if process-unlock is called on it when it is not locked by the current process, and potentially in other ‘dangerous’ circumstances. An unsafe lock does not signal these errors. The default value of safep is t.

recursivep, when true, allows lock to be locked recursively. Trying to lock a lock that is already locked by the current thread just increments its lock count. If lock is created with recursivep nil then trying to lock again causes an error. This is useful for debugging code where lock is never expected to be claimed recursively. The default value of recursivep is t.

sharing, when true, causes lock to be a "sharing" lock object, which supports sharing and exclusive locking. At any given time, a sharing lock may be free, or it may be locked for sharing by any number of threads or locked for exclusive use by a single thread. Sharing locks are handled by different functions and methods from non-sharing locks.

Example

CL-USER 3 > (setq *my-lock* (mp:make-lock :name "my-lock"))
#<MP:LOCK "my-lock" Unlocked 2008CAC7>

CL-USER 4 > (mp:process-lock *my-lock*)
T

CL-USER 5 > *my-lock*
#<MP:LOCK "my-lock" Locked once by "CAPI Execution Listener 1" 2008CAC7>

CL-USER 6 > (mp:process-lock *my-lock*)
T

CL-USER 7 > *my-lock*
#<MP:LOCK "my-lock" Locked 2 times by "CAPI Execution Listener 1" 2008CAC7>
**function**

**make-mailbox**

**Summary** Makes a new mailbox.

**Package** mp

**Signature** make-mailbox &key size name => mailbox

**Arguments**

- size An integer
- name A Lisp object

**Values** mailbox A mailbox

**Description** The function make-mailbox returns a new mailbox.

size specifies the initial size of the mailbox mailbox.

The reader process is set to nil.

name does not affect the functionality of mailbox, but can be very useful for debugging. It appears in the printed representation of mailbox, and also in the process-whostate of any process that waits for mailbox (using mailbox-read).

**See also**

- mailbox
- mailbox-empty-p
- mailbox-peek
- mailbox-read
make-named-timer

Function

Summary
Creates and returns a named timer.

Package
mp

Signature
make-named-timer name function &rest arguments => timer

Arguments
name A string or symbol
function A function
arguments A set of arguments to function

Values
timer A timer

Description
The function make-named-timer creates and returns a named timer. The first argument is a string or symbol naming the timer. The second argument is a function to be applied to the remaining arguments when the timer expires. Use the function schedule-timer or schedule-timer-relative to set an expiration time.

In comparison, the function make-timer creates an unnamed timer.

Example
(setq timer (mp:make-named-timer 'timer-1 'print 10 *standard-output*))

#<Time Event TIMER-1 : PRINT>
See also
  make-timer
  schedule-timer
  schedule-timer-milliseconds
  schedule-timer-relative
  schedule-timer-relative-milliseconds
  timer-expired-p
  timer-name
  unschedule-timer

make-semaphore

Function

Summary
Makes a semaphore.

Package
mp

Signature
make-semaphore &key name count => sem

Arguments
name An object.
count A non-negative fixnum.

Values
sem A semaphore.

Description
The function make-semaphore returns a new semaphore for use with semaphore-acquire and semaphore-release. The unit count is initialized to count, which defaults to 1. If name is supplied, the semaphore will have that name. If name is not supplied, the semaphore will be given a unique anonymous name.

See also
  semaphore
  semaphore-acquire
  semaphore-count
  semaphore-name
  semaphore-release
  semaphore-wait-count
  “Counting semaphores” on page 287
The function `make-timer` creates and returns an unnamed timer. The `function` argument is a function to be applied to the remaining arguments when the timer expires. Use the function `schedule-timer` or `schedule-timer-relative` to set an expiration time.

If `function` returns the keyword `:stop`, then `timer` is unscheduled (as if by `unschedule-timer`). This allows you to schedule a repeating timer (by passing `repeat-time` to `schedule-timer`, `schedule-timer-relative`, `schedule-timer-milliseconds` or `schedule-timer-relative-milliseconds`) that unschedules itself when some condition is true. Otherwise the values returned by `function` are ignored.

Note that the function `make-named-timer` creates a named timer.

```
(setq timer
  (mp:make-timer 'print 10 *standard-output*))
=>
#<Time Event : PRINT>
```

See also

- `make-named-timer`
- `make-timer`
- `schedule-timer`
- `schedule-timer-milliseconds`
map-all-processes

**Function**

**Summary**: Calls a predicate function on processes in turn until a true value is returned.

**Package**: mp

**Signature**: `map-all-processes function => result`

**Arguments**

- `function`: A function taking one argument

**Values**

- `result`: A process or `nil`.

**Description**: The function `map-all-processes` calls `function` on processes.

`function` is passed each process in turn as its single argument.

For a process argument `p`, if `function` returns `nil` then `map-all-processes` continues by calling `function` on the next process, but if `function` returns true then `map-all-processes` returns `p` immediately and stops calling `function` (so `function` may not get called on all processes).

**See also**: `map-processes`

map-all-processes-backtrace

**Function**

**Summary**: Produces a backtrace for every known process.
The function map-all-processes-backtrace calls function, which defaults to print, for every known process and each line of its backtrace.

See also map-process-backtrace

map-process-backtrace

Function

Summary Produces a backtrace for a process

Package mp

Signature map-process-backtrace process function

Arguments process A process

function A function taking one argument

Values None.

Description The function map-process-backtrace collects a backtrace for the process specified by process, and the function function is called on each line of the backtrace in turn.
Example

CL-USER 1 > (mp:map-process-backtrace mp:*current-process* 'print)

DBG::GET-CALL-FRAME
MP:MAP-PROCESS-BACKTRACE
SYSTEM::%INVOKE
SYSTEM::%EVAL
EVAL
SYSTEM::DO-EVALUATION
SYSTEM::%TOP-LEVEL-INTERNAL
SYSTEM::%TOP-LEVEL
SYSTEM::LISTENER-TOP-LEVEL
CAPI::CAPI-TOP-LEVEL-FUNCTION
CAPI::INTERACTIVE-PANE-TOP-LOOP
(SUBFUNCTION MP::PROCESS-SG-FUNCTION MP::INITIALIZE-PROCESS-STACK)
SYSTEM::%FIRST-CALL-TO-STACK
NIL

See also map-all-processes-backtrace

map-processes

Function

Summary
Calls a predicate function on processes in turn until a true value is returned.

Package
mp

Signature
map-processes function => result

Arguments
function A function taking one argument

Values
result A process or nil.

Description
The function map-processes calls function on processes.

function is passed each live process (as determined by process-alive-p) in turn as its single argument.

For a process argument p, if function returns nil then map-processes continues by calling function on another process,
but if function returns true then \texttt{map-processes} returns \texttt{p} immediately and stops calling \texttt{function} (so \texttt{function} may not get called on all processes).

See also \texttt{map-all-processes}

\textbf{notice-fd} \hfill \textit{Function}
\begin{description}
\item[	extbf{Summary}] Add a file descriptor to the set of interesting input file descriptors.
\item[	extbf{Package}] \texttt{mp}
\item[	extbf{Signature}] \texttt{notice-fd} \texttt{fd}
\item[	extbf{Arguments}] \texttt{fd} \hfill A UNIX file descriptor
\item[	extbf{Values}] None.
\item[	extbf{Description}] The function \texttt{notice-fd} adds the given \texttt{fd} to the set of fds that cause LispWorks to wake up when they contain input. This function is not implemented on Microsoft Windows.
\item[	extbf{See also}] \texttt{unnotice-fd}
\end{description}

\textbf{process-alive-p} \hfill \textit{Function}
\begin{description}
\item[	extbf{Summary}] Determines if a process is alive.
\item[	extbf{Package}] \texttt{mp}
\item[	extbf{Signature}] \texttt{process-alive-p} \texttt{process} \Rightarrow \texttt{bool}
\item[	extbf{Arguments}] \texttt{process} \hfill A process
\end{description}
Values  

| bool | A boolean |

Description  
The function `process-alive-p` returns `t` if `process` is alive, that is, if `process-terminate` has not been called on the process.

Example  

```lisp
(mp:process-alive-p mp:*current-process*)
=> T
(let ((process (mp:process-run-function "test" nil 'identity nil)))
  (sleep 2)
  (mp:process-alive-p process))
=> NIL
```

`process-all-events`  

Function

Summary  
Processes the events in the mailbox of the current process.

Package  
`mp`

Signature  
`process-all-events => processedp`

Values  

| processedp | A boolean. |

Description  
The function `process-all-events` processes all the events in the mailbox of the current process, by calling `general-handle-event` on each one of them.

`process-all-events` returns a boolean indicating whether it processed any event.

See also  
`general-handle-event`
`process-mailbox`
`process-send`
**process-allow-scheduling**

**Function**

Summary: Allows scheduling within a process, so that the process is interruptible.

Package: mp

Signature: `process-allow-scheduling =>`

Arguments: None.

Values: None.

Description: This gives other Lisp processes a chance to run.

**process-arrest-reasons**

**Function**

Summary: Returns a list of the reasons why a Lisp process has stopped.

Package: mp

Signature: `process-arrest-reasons process => reasons`

Arguments: `process` A process.

Values: `reasons` A list of reasons.

Description: The function `process-arrest-reasons` returns a list of the reasons why a Lisp process has stopped. A process is inactive if it has any arrest reasons.

Use of `(setf mp:process-arrest-reasons)` is deprecated. You should use `process-stop` instead. If you set the arrest reasons of the current process, this causes the current process to stop immediately, before returning from `mp:process-arrest-reasons` (like `process-stop`).
Compatibility notes: The immediate stopping behavior of `(setq mp:process-arrest-reasons)` is different from LispWorks 5.0 and previous versions.

See also: `process-run-reasons` `process-stop`

### process-break

**Function**

**Summary:** Breaks a Lisp process and enters the debugger.

**Package:** `mp`

**Signature:** `process-break process =>`

**Arguments:**

- `process`: A process.

**Values:** None.

**Description:** The function `process-break` forces the process `process` to break and enter the debugger.

See also: `debug-other-process`

### process-continue

**Function**

**Summary:** Wakes up a process.

**Package:** `mp`

**Signature:** `process-continue process => nil`

**Arguments:**

- `process`: A `mp:process` object.
The function `process-continue` wakes up the process `process`, regardless of whether it is sleeping, stopped or waiting.

`process-continue` returns `nil`.

### process-exclusive-lock

**Function**

**Summary**
Like `process-lock`, but on a "sharing" lock.

**Package**
mp

**Signature**
`process-exclusive-lock sharing-lock &optional whostate timeout`

**Arguments**
- `sharing-lock` A sharing lock.
- `whostate` The status of the process while lock is locked, as seen in the Process Browser.
- `timeout` A non-negative real or `nil`.

**Description**
The function `process-exclusive-lock` is the same as `process-lock`, but on a "sharing" lock. It waits until `sharing-lock` is free before locking it in exclusive mode.

Calls to `process-exclusive-lock` should be paired with `process-exclusive-unlock` calls. In most cases the macro `with-exclusive-lock` the best way to achieve this.

**Notes**
It is not possible to use exclusive lock in the scope of a sharing lock on the same lock, and trying to do this will cause the process to hang. Whether it is possible to use an exclusive lock inside an exclusive lock of the same lock is determined by the `recursivep` argument in `make-lock`.

`process-exclusive-lock` is guaranteed to return if it locked `sharing-lock`, but may throw before locking, as
described in “Guarantees and limitations when locking and unlocking” on page 278.

See also
make-lock
process-lock
with-exclusive-lock
“Locks” on page 277

process-exclusive-unlock

Summary Like process-unlock, but on a "sharing" lock.

Package mp

Signature process-exclusive-unlock sharing-lock

Arguments sharing-lock A sharing lock.

Description The function process-exclusive-unlock is the same as process-unlock but for a "sharing" lock.

Calls to process-exclusive-unlock should be paired with process-exclusive-lock calls. In most cases the macro with-exclusive-lock the best way to achieve this.

Notes process-exclusive-unlock is guaranteed to successfully unlock sharing-lock, but is not guaranteed to return, as described in “Guarantees and limitations when locking and unlocking” on page 278.

See also process-exclusive-lock
process-unlock
with-exclusive-lock
“Locks” on page 277
**process-idle-time**

*Function*

**Summary**
Returns the time for which a process has been idle.

**Package**
mp

**Signature**
process-idle-time process => time

**Arguments**
process A process.

**Values**
time A non-negative integer.

**Description**
The function process-idle-time returns the length of time in internal time units that process has been idle. If the process is running (for example the current process) then the return value is 0.

**See also**
process-run-time

**process-initial-bindings**

*Variable*

**Summary**
Specifies the variables initially bound in a new process.

**Package**
mp

**Description**
This specifies the variables that are initially bound in a Lisp process when that process is created. This variable is an association list of symbols and initial value forms. The initial value forms are processed by a simple evaluation that handles symbols and function call forms, but not special operators. As a special case, if the value form is the same as the symbol and that symbol is unbound, then the symbol will be unbound in the new process.

When process-run-function is called, it performs the evaluation in the dynamic environment of the call. It is therefore possible to bind *process-initial-bindings* dynamically.
cally around the call, as shown in the examples below, and that is the preferred way of using *process-initial-bindings* (rather than setting the global value).

The initial value forms in *process-initial-bindings* are also evaluated in outer calls on foreign threads, which are threads that were made by foreign code rather than by Lisp. See “Foreign callbacks on threads not created by Lisp” on page 297 for discussion. Note that in this case, the evaluation occurs in a dynamic environment where the variables in *process-initial-bindings* are not bound yet. That is different from calls to process-run-function, where all the variables that were in *process-initial-bindings* at the time that the calling process was created are already bound.

**Notes**

Changing the global value of *process-initial-bindings* affects all new processes in the system, including processes that will be associated with foreign threads. Unless that is what you want, you should not set the global value. When you do set it, you should take care to avoid errors.

Errors in the evaluation are signaled in the standard way when they occur due to process-run-function. When LispWorks creates processes for its own use, it just ignores such errors and binds the corresponding variable to nil. When the evaluation is for a foreign thread, the error is signaled as usual, wrapped with a restart that allows you to use nil as the value.

**Examples**

This example shows a typical use with a bound symbol:

```lisp
(defvar *binding-1* 10)
```
(let ((mp:*process-initial-bindings*  
      (cons '(*binding-1* . 20)  
        mp:*process-initial-bindings*)))  
  (mp:process-run-function  
    "binding-1"  
    ()  
    #'(lambda (stream)  
        (format stream "&Binding 1 is ~S.~%" *binding-1*))  
    *standard-output*)  
  (sleep 1))  
=>  
Binding 1 is 20.

This example shows the special case with an unbound symbol:

(defvar *binding-2*)

(let ((mp:*process-initial-bindings*  
      (cons '(*binding-2* . *binding-2*)  
        mp:*process-initial-bindings*)))  
  (flet ((check-binding-2 ()  
        (mp:process-run-function  
          "binding-2"  
          ()  
          #'(lambda (stream)  
              (if (boundp '*binding-2*)  
                  (format stream "&Binding 2 is ~S.~%"  
                      *binding-2*)  
                  (format stream "&Binding 2 is unbound.~%"))  
          *standard-output*)  
        (sleep 1))))  
  (check-binding-2)  
  (let ((*binding-2* 123))  
    (check-binding-2)))  
=>  
Binding 2 is unbound.  
Binding 2 is 123.

**process-internal-server-p**

*Function*

**Summary** Tests whether a process is an internal server.
**Package**  
`mp`

**Signature**  
`process-internal-server-p process => result`

**Arguments**  
`process`  
A `mp:process` object.

**Values**  
`result`  
A boolean.

**Description**  
The function `process-internal-server-p` is the predicate for whether a process is marked as "internal server".

**Notes**  
Processes are marked as "internal server" by a true value for `:internal-server` amongst the `keywords` in a call to `process-run-function`.

**See also**  
`process-run-function`  
`any-other-process-non-internal-server-p`

---

**process-interrupt**  

*Function*

**Summary**  
Interrupts a process.

**Package**  
`mp`

**Signature**  
`process-interrupt process function &rest arguments =>`

**Arguments**  
`process`  
A process.

`function`  
A function to apply on resuming `process`.

`arguments`  
Arguments to supply to `function`.

**Values**  
None.

**Description**  
The function `process-interrupt` causes the Lisp process `process` to apply `function` to `arguments` when it is next resumed. Afterwards the process resumes its normal execution, as long
as function does not throw. A waiting process is temporarily woken up.

Notes

Interrupts should be used only for simple operations such as setting a variable. Any more complex interrupt function is potentially dangerous and should be avoided. The problem is that even simple code like:

\[
\text{(let ((message (read-message)))}
\text{(process-message message))}
\]

may lose the message if an interrupt ends up throwing between the two lines. In addition, the code in the interrupt may be executed while some tree of pointers is in an inconsistent state (while the message is incompletely processed, for example).

See also process-interrupt-list

**process-interrupt-list**  
*Function*

Summary  
Interrupts a process.

Package  
mp

Signature  
process-interrupt-list process function arguments =>

Arguments  
process  
A process.

function  
A function to apply on resuming process.

arguments  
A list of the arguments to supply to function.

Values  
None.

Description  
The function process-interrupt-list causes the Lisp process process to apply function to arguments when it is next
resumed. It is just like **process-interrupt** except that the arguments are supplied as a list.

See also  

**process-interrupt**

---

**process-join** **Function**

**Summary**  
Waits until a specified process dies, or a *timeout* is reached.

**Package**  
mp

**Signature**  
`process-join process &key timeout => flag`

**Arguments**  
- `process`  
  A process.
- `timeout`  
  A non-negative **real** or **nil**.

**Values**  
- `flag`  
  A boolean.

**Description**  
The function **process-join** waits until the process `process` dies, or `timeout` seconds passed.

If the process dies then **process-join** returns **t**. If the timeout passed it returns **nil**.

**process-join** can be used on dead processes, and in this case returns **t** immediately.

The effect of **process-join** is similar to

```
(mp:process-wait-with-timeout  
 "Waiting for process to die" timeout  
 #'(lambda (x)  
   (not (mp:process-alive-p x))) process)
```

but the call above may not return until the next time the scheduler runs, possibly causing a delay. In contrast **process-join** returns immediately when the process dies.

See also  

**process-wait-with-timeout**
process-kill

Summary
Kills the specified Lisp process. process-kill is deprecated.

Package
mp

Signature
process-kill process =>

Arguments
process A process.

Values
None.

Description
The function process-kill kills the specified Lisp process.

1. process-kill kills the process by sending it an interrupt with current-process-kill, which will throw out of whatever it is doing. That means that any code that is executing without interrupts blocked may abort in the middle. It is wise in general to block interrupts around all sensitive places, so that process-kill may kill the process in a non-sensitive place.

2. If process-kill is called while the process is in a no-interrupt context, the killing will actually happen when the process exits that no-interrupt context.

3. If the killing happens inside the cleanup forms of unwind-protect, it may terminate a cleanup in the middle. It is possible to protect against this by doing all cleanups with interrupts disallowed, but that is not easy. Thus process-kill may be problematic, and should be avoided when possible. Whenever possible, make your processes check some flag that can be set by other threads and exit when the flag is set to some value.
See also  
ensure-process-cleanup  
process-terminate  

**process-lock**  

*Function*

**Summary**

Locks a **lock** for the current process.

**Package**

**mp**

**Signature**

`process-lock lock &optional whostate timeout => result`

**Arguments**

- **lock**  
  A **lock**.

- **whostate**  
  The status of the current Lisp process, before *process-lock* returns, that is, the status while the current process is waiting to time-out. This can be seen in the Process Browser.

- **timeout**  
  A timeout interval, in seconds. If this is **nil** (the default), *process-lock* waits until **lock** can be locked by the current Lisp process. A process can lock a **lock** more than once.

**Values**

- **result**  
  A boolean.

**Description**

The function **process-lock** attempts to lock **lock** and returns **t** if successful, or **nil** if timed out.

If **lock** is already locked and its owner is the value of *current-process*, then the value of Recursivep in the **lock** (see `make-lock`) controls what happens. If **recursivep** is true, then **lock** remains locked and an internal count is incremented (this is called recursive locking). Otherwise, an error is signaled.

The Lisp process sleeps until the lock can be locked or the timeout period specified by **timeout** expires.

**result** is **t** if **lock** was successfully locked, and **nil** otherwise.
Notes  process-lock is guaranteed to return if it locked process, but may throw before locking, as described in “Guarantees and limitations when locking and unlocking” on page 278.

Example  (process-lock *my-lock* "waiting to lock" 10)

See also  make-lock
         process-exclusive-lock
         process-unlock
         with-lock
         “Locks” on page 277

process-mailbox

Function

Summary  Accesses the mailbox associated with a process.

Package  mp

Signature  process-mailbox process => result

(setf process-mailbox) result process => result

Arguments  process  A process.

Values  result  A mailbox or nil.

Description  process-mailbox is an accessor function which returns or sets the mailbox associated with process.

Example  (setf (mp:process-mailbox mp:*current-process*)
              (mp:make-mailbox))

process-name

Function

Summary  Returns the name of a specified process.
**Package**  
mp

**Signature**  
process-name process => name

**Arguments**  
process A process.

**Values**  
name The name of the process specified by process.

**Description**  
The function **process-name** returns the name of the specified Lisp process.

---

**process-p**  
*Function*

**Summary**  
The predicate for processes.

**Package**  
mp

**Signature**  
process-p object => bool

**Arguments**  
object Any object

**Values**  
bool A generalized boolean.

**Description**  
The function **process-p** returns t if object is a process, and nil otherwise.

---

**process-plist**  
*Function*

**Summary**  
Returns the plist associated with a process. This function is deprecated.

**Package**  
mp

**Signature**  
process-plist process => plist
Arguments

process  A process

Values

plist  A plist

Description

The function process-plist returns the plist associated with process.

Notes

It is not possible to manipulate the plist in a thread-safe manner, and process-plist may interact badly with other users of the plist, hence process-plist is deprecated. Use instead process-property and get-process-private-property etc.

process-poke

Function

Summary

Makes a waiting process call its wait function.

Package

mp

Signature

process-poke process => result

Arguments

process  A process.

Values

result  A boolean.

Description

If the process process is waiting, the function process-poke causes it to run its wait-function as soon as possible, and if the wait function returns true, the process returns from the waiting function.

process-poke is especially useful when using the process-wait-local-* functions. With process-wait-local and process-wait-local-with-timeout, it is the only way to ensure that the waiting process checks the wait function. The other functions also check periodically, but process-poke is still useful to make them wake up immediately.
With a non-local wait function (that is, in `process-wait` and `process-wait-with-timeout`), `process-poke` is useful in SMP LispWorks to ensure that the process wakes and checks its `wait-function` immediately. `process-poke` has no effect on non-SMP LispWorks for `process-wait` and `process-wait-with-timeout`.

You can also use `process-poke` to wake up a process that waits using `current-process-pause`.

`process-poke` returns `t` if it actually poked the process or `nil` otherwise (when the process is not waiting or is stopped).

The process wait functions are designed to call the `wait-function` just before going to sleep, in a way that guards against a race condition between `process-poke` and the waiting function. In particular, they ensure that if a process goes to wait with a `wait-function` that checks some value, and another process sets this value and calls `process-poke` on the first process, the first either will check the value before going to sleep, or wake up and check the value. The first process is never going to get stuck because it went to sleep just as the other process set the value. Note that this is guaranteed only when the value is set before `process-poke` is called.

Functions that cause specific wait functions to be ready to run (for example `mailbox-send` which causes `mailbox-read` to be ready to run) implicitly pokes a process that waits, so there is no need to use `process-poke` when these functions are used.

**Example**

Worker process function:

```lisp
(defun worker-process-function (work-struct)
  (loop (mp:process-wait-local "Waiting for request" 'worker-struct-request work-struct)
        (process-request 'worker-struct-request work-struct)
        (setf (worker-struct-request work-struct) nil)))
```
Another process distributes requests:

```
(dolist (work-struct *work-structs*)
  (unless (worker-struct-request work-struct)
    (setf (worker-struct-request work-struct) request)
    (mp:process-poke
      (worker-struct-process work-struct))
    (return work-struct)))
```

This specific example can be implemented a little more simply by `mailbox-read` and `mailbox-send`, but if the wait function needs to check for something else it can be easily added.

See also
- `current-process-pause`
- `process-wait`
- `process-wait-local`
- `process-wait-local-with-periodic-checks`
- `process-wait-local-with-timeout`
- `process-wait-local-with-timeout-and-periodic-checks`
- `process-wait-with-timeout`

### process-priority

**Function**

**Summary**

Returns the numerical priority of the Lisp process.

**Package**

`mp`

**Signature**

`process-priority process => priority`

**Arguments**

- `process`: A process.

**Values**

- `priority`: A fixnum, the priority of `process`.

**Description**

Returns the numerical priority of the Lisp process. This can be modified by calling `change-process-priority`.
Example

CL-USER 17 > (mp:process-priority mp::*current-process*)
600000

See also change-process-priority

process-private-property

Function

Summary

Gets or sets the value of a private property of the current process.

Package

mp

Signature

process-private-property indicator &optional default => result

(setq process-private-property) value indicator &optional default => result

Arguments

indicator A Lisp object.

default A Lisp object.

Values

result value or default

Description

The function process-private-property gets or sets the value that is associated with indicator in the private properties of the current process (that is, the result of calling get-current-process).

If indicator is not associated with a value in the private properties, process-private-property returns default.

(setq process-private-property) overwrites any existing value for indicator.

The default value of default is nil.

Notes

1. Private properties can be read from other processes using get-process-private-property, but cannot be set by other processes.
2. Process private property access is faster than process property access in SMP LispWorks, because the implementation of the latter must deal with parallel setting.

See also

- remove-process-private-property
- pushnew-to-process-private-property
- remove-from-process-private-property
- get-process-private-property

### process-property

**Function**

**Summary**

Gets and sets a general property for a process.

**Package**

mp

**Signature**

```lisp
process-property indicator &optional process default => result
(setf process-property) value indicator &optional process
default => result
```

**Arguments**

- `indicator` A Lisp object.
- `process` A process.
- `default` A Lisp object.

**Values**

- `result` A property value, or default.

**Description**

The function `process-property` gets the value that is associated with `indicator` for the process `process`, and `(setf process-property)` sets this value.

If `process` is not supplied or is `nil`, the current process (that is, the result of calling `get-current-process`) is used.

**Notes**

In the typical case when only the current process sets the property (even if other processes read it), private properties can be used, and are much faster in SMP LispWorks, because
they do not need to deal with parallel setting. See \texttt{process-private-property}.

Example

\begin{verbatim}
(process-property 'foo (get-current-process) 'bar)
=> BAR
(setf (process-property 'foo) 'foo-value)
=> FOO-VALUE
(process-property 'foo)
=> FOO-VALUE
\end{verbatim}

See also \texttt{process-private-property remove-process-property remove-from-process-property pushnew-to-process-property}

\textbf{process-reset} \textit{Function}

Summary

Resets a process by discarding its current state.

Package \texttt{mp}

Signature

\texttt{process-reset process =>}

Arguments

\begin{verbatim}
process A process.
\end{verbatim}

Values

None.

Description

\texttt{process-reset} interrupts the execution of process and “throws away” its current state. Upon resuming execution, the process calls its function with its initial argument and priority.

\texttt{process-reset} modifies the dynamic execution state of \texttt{process}. It performs a non-local exit from the currently run-
ning function, to cause the process’s main function to return. 
\texttt{unwind-protect} forms will be run.

\texttt{process-reset} does not modify any of the attributes of the process, in particular its priority, items on the plist, or accumulated run-time.

Notes Since \texttt{process-reset} causes an asynchronous non-local exit, it is possible that it can occur within an \texttt{unwind-protect} cleanup form or before data used by an \texttt{unwind-protect} cleanup form has been initialized. In some cases, not all cleanups within that form will be run.

\begin{description}
\item[\texttt{process-run-function}]
\begin{description}
\item[Summary] Create a new process, passing it a function to run.
\item[Package] \texttt{mp}
\item[Signature] \texttt{process-run-function name keywords function \&rest arguments => process}
\item[Arguments]
\begin{itemize}
\item \texttt{name} A name for the new process.
\item \texttt{keywords} Keywords specifying properties of the new process.
\item \texttt{function} A function to apply.
\item \texttt{arguments} Arguments to pass to \texttt{function}.
\end{itemize}
\item[Values] \texttt{process} The newly created process.
\item[Description] This function creates a new Lisp process with name \texttt{name}. Other properties of \texttt{process} may be specified in keyword/value pairs in \texttt{keywords}:
\end{description}
\end{description}
:priority A fixnum representing the priority for the process. If \texttt{priority} is not supplied, the process priority becomes the value of the variable \texttt{*default-process-priority*}.

:mailbox A mailbox, a string, \texttt{t} or \texttt{nil}, used to initialize the \texttt{process-mailbox} of \texttt{process}.

True values specify that \texttt{process} should have a mailbox. A \texttt{mailbox} is used as-is; a string is used as the name of a new mailbox; and \texttt{t} causes it to create a \texttt{mailbox} with the same name as \texttt{process}, that is, \texttt{name}.

Note that both \texttt{process-send} and \texttt{process-wait-for-event} force the relevant process to have a \texttt{mailbox}.

:internal-server

When true, this indicates that the process is an "internal server", which means that it responds to requests for work from other processes. The main effect of this is that if the only processes that remain are "internal servers", nothing is going to happen, so LispWorks quits. The system marks some of the processes that it creates as "internal server".

:remote-terminator

A function designator for a function of one argument.

:local-terminator

A function designator for a function of no arguments.

:terminate-by-send

A generalized boolean.
The new process is preset to apply function to arguments and runs in parallel, while process-run-function returns immediately.

:remote-terminator, :local-terminator or :terminate-by-send define the Terminate Method of the process, which is what process-terminate uses. If more than one of these keyword arguments is supplied, then :remote-terminator takes precedence over :local-terminator which takes precedence over :terminate-by-send.

If remote-terminator is supplied, it must be a function of one argument. When process-terminate is called, it funcalls remote-terminator on the process that process-terminate was called on, which normally will be another process. It should then terminate the process somehow. Typically the process itself will be frequently checking some flag which tells it to exit, and the function remote-terminator just sets this flag. remote-terminator should return non-nil when it is "successful", that is it did something that should cause the process to terminate. process-terminate checks the result of the call to remote-terminator, and if it is nil it also calls process-kill on the process.

If local-terminator is supplied, it must be a function of no arguments. When process-terminate is called it sends to the process a list with the local-terminator as the only element. That relies on the process itself processing what is sent to it and funcalling the function. This is what general-handle-event does, which is what system processes tend to use. In particular, all processes that are created by CAPI use it.

If terminate-by-send is supplied and non-nil, process-terminate sends the process a list containing current-process-kill (that is it is the same as :local-terminator 'current-process-kill). CAPI processes use this keyword.
Example

Example

```
CL-USER 253 > (defvar *stream* *standard-output*)
*STREAM*
CL-USER 254 > (mp:process-run-function
    "My process"
    '(:priority 42)
    #'(lambda (x)
        (loop for i below x
            do (and (print i *stream*)
               (sleep 1))
        finally
        (print (mp:process-priority
               mp:*current-process*)
               *stream*)))
3)
#<MP:PROCESS Name "My process" Priority 850000 State "Running">
0
1
2
42
CL-USER 255 >
```

See also

current-process-kill
*default-process-priority*
*initial-processes*
list-all-processes
map-processes
process-alive-p
process-join
process-terminate
process-whostate
ps

---

**process-run-reasons**

Function

Summary

Returns the reasons that a specified process is running.

Package

mp
**process-run-reasons**

**Signature**

\[ \text{process-run-reasons \ process =\> reasons} \]

\[(\text{setf process-run-reasons}) \\ \text{process reasons =\> reasons} \]

**Arguments**

- `process`: A process.

**Values**

- `reasons`: A list of run reasons.

**Description**

The function `process-run-reasons` returns a list of reasons for the specified Lisp process running. These can be changed using `setf`.

A process is only active if it has at least one run reason and no arrest reasons.

**See also**

- `process-arrest-reasons`
- `process-run-function`
- `process-whostate`

---

**process-run-time**

**Function**

**Summary**

Returns the current run time for a process.

**Package**

`mp`

**Signature**

\[ \text{process-run-time \ process =\> time} \]

**Arguments**

- `process`: A process.

**Values**

- `time`: A positive integer or `nil`.

**Description**

The function `process-run-time` returns the current run time for `process` in internal time units. If the value cannot be determined (currently this is only on FreeBSD), then the return value is `nil`. 

---
Notes

The value returned by `get-internal-run-time` is similar, but on some operating systems it is the total time for all Lisp processes in the image.

See also  

`process-idle-time`

---

**process-send**

*Function*

**Summary**

Sends an object to the mailbox of a given process.

**Package**

`mp`

**Signature**

```
process-send process object &key change-priority limit timeout error-if-dead-p => success
```

**Arguments**

- `process` A process
- `object` An object
- `change-priority` A fixnum, `nil`, `t`, or `:default`
- `limit` An integer or `nil`
- `timeout` A non-negative `real` or `nil`
- `error-if-dead-p` A generalized boolean.

**Values**

- `success` A boolean.

**Description**

The function `process-send` queues `object` in the mailbox of the given process.

`object` can any kind of Lisp object, and it is up to the receiving process to interpret it.

`process-send` only sends the event: it is the responsibility of the receiving process to actually read the event and then interpret it. Reading is typically done by calling `process-wait-for-event`. Interpreting the event is up the caller of `process-wait-for-event`. In the "standard" situation, for
example in a process started by CAPI, the object will be processed as an event by calling `general-handle-event`.

`process-send` actually uses the `process-mailbox` of `process`, creating a mailbox for `process` if it does not already have one. In principle `object` can be read by another process, by calling `mailbox-read` (or `process-wait-for-event`) on the mailbox.

If `change-priority`, which has a default value of `:default`, is non-nil, it controls how the priority of that process is calculated as follows:

- `fixnum` — use the value of `change-priority` as the new priority.
- `t` — set the priority to the interactive priority.
- `:default` — set the priority to the normal running priority.

`error-if-dead-p` defaults to `nil`, which means that if `process-send` is called with a dead process, it just returns false. If `error-if-dead-p` is non-nil, when `process-send` is called on a dead process it signals a continuable error.

`limit` defaults to `nil`. If it is non-nil, it must be a positive integer that specifies the maximum size to which `process-send` may expand the mailbox of the process. When `limit` is non-nil, `process-send` adds the object to the mailbox as if by

`(mailbox-send-limited mailbox object limit timeout)`

See `mailbox-send-limited` for details.

`timeout` defaults to `nil` and is used when `limit` is non-nil as described above, otherwise it is ignored.

`process-send` returns true if it put the object in the mailbox of the process and false otherwise. The latter can happen either because the process is dead, or because the process’s mailbox is full and reached the size specified by `limit` and `timeout` is non-nil.
See also

general-handle-event
mailbox-send
mailbox-send-limited
process-wait-for-event
“Communication between processes and synchronization” on page 283

process-sharing-lock

Function

Summary Like process-lock, but on a "sharing" lock.

Package mp

Signature process-sharing-lock sharing-lock &optional whostate timeout

Arguments

sharing-lock A sharing lock.

whostate The status of the process while lock is locked, as seen in the Process Browser.

timeout A non-negative real or nil.

Description This is like process-lock, but sharing-lock must be a "sharing" lock and it will be locked in shared mode. That means that other threads can also lock it in shared mode.

Before locking, process-sharing-lock waits for sharing-lock to be free of any exclusive lock, but it does not check for other shared mode use of the same lock.

Calls to process-sharing-lock should be matched by calls to process-sharing-unlock with sharing-lock. Normally with-sharing-lock is the best way to achieve this.

Notes It is possible to lock for sharing inside the scope of a sharing lock and inside the scope of an exclusive lock.

process-sharing-lock is guaranteed to return if it locked sharing-lock, but may throw before locking, as described in
“Guarantees and limitations when locking and unlocking” on page 278.

See also

process-lock
process-sharing-unlock
with-sharing-lock
“Locks” on page 277

**process-sharing-unlock**  

*Function*

**Summary**  
Removes a sharing lock.

**Package**  
mp

**Signature**  
process-sharing-unlock sharing-lock

**Arguments**  
sharing-lock  
A sharing lock.

**Description**  
The function process-sharing-unlock is the same as process-unlock but for a "sharing" lock.

Calls to process-sharing-unlock should be matched by calls to process-sharing-lock with sharing-lock. Normally with-sharing-lock is the best way to achieve this.

**Notes**  
process-sharing-unlock is guaranteed to successfully unlock sharing-lock, but is not guaranteed to return, as described in “Guarantees and limitations when locking and unlocking” on page 278.

See also

process-unlock
with-sharing-lock
“Locks” on page 277
**Function**

**process-stop**

**Summary**
Stops a process.

**Package**
mp

**Signature**
process-stop process

**Arguments**
process A mp:process object.

**Description**
The function process-stop stops the process process.

process must be a full process (that is, not one created by *current-process*).

process-stop causes process to stop until some other process explicitly wakes it up. If it is called on the current process, the current process stops during the call, and returns from process-stop after the process gets woken up.

In SMP LispWorks, if process is not the current process, process-stop returns immediately and the execution of process stops at some point, possibly after process-stop returned. In non-SMP LispWorks if process is not the current process, process stops before process-stop returns.

You can wake up a stopped process (that is, make it runnable) by calling process-terminate, process-unstop or process-continue.

process-interrupt does not wake up a stopped process.

There is a discussion of a typical use of process-stop in the section “Stopping and unstopping processes” on page 296.

process-stop does not return any useful value.

**See also**
process-arrest-reasons
process-stopped-p
process-unstop
process-stopped-p

Function

Summary The predicate for stopped processes.

Package mp

Signature process-stopped-p process => result

Arguments process A mp:process object.

Values result A boolean.

Description The function process-stopped-p queries whether the process process is stopped or not.

If process stopped because it called process-stop on itself, then process-stopped-p result is t only if process-stop really stopped it (that is, a later call to process-unstop will unstop the process).

See also process-stop process-unstop

process-terminate

Function

Summary Kills a process "nicely".

Package mp

Signature process-terminate process &key join-timeout force-timeout

Arguments process A mp:process object.

join-timeout A non-negative real or nil.

force-timeout A non-negative real or nil.
The function `process-terminate` terminates the process `process`, which means killing it "nicely". `process-terminate` invokes the Terminate Method of `process`, if it has one, otherwise it calls `process-kill`.

The terminate is set either by supplying one of `local-terminator`, `remote-terminator` or `terminate-by-send` in the call to `process-run-function`, or by a call to `current-process-set-terminate-method` on the process. See the entry for `process-run-function` for details.

If the process does not have a Terminate Method, `process-terminate` calls `process-kill`.

If `force-timeout` is non-nil then `process-terminate` sets a timer that kills the process after `force-timeout` seconds.

If `join-timeout` is non-nil then it is the time in seconds to "join" the process, that is waiting for it to die. When `join-timeout` is non-nil, after invoking the Terminate Method or calling `process-kill`, `process-terminate` calls `process-join` using `join-timeout` as the timeout, and returns the result.

`process-terminate` returns the result of `process-join` if `join-timeout` is non-nil, otherwise it returns 0.

Notes

1. `process-terminate` is the appropriate way to kill processes, because it gives the process the option to decide when to exit. `process-kill` kills the process whenever it is not blocking interrupts, which may still be sensitive in some sense.

2. When multiprocessing stops (for example when quitting, or saving a session), the system uses first `process-terminate` and then `process-kill`, so processes that exit with `process-terminate` have the chance to clean up as needed.
3. `process-terminate` is better than `process-kill` only when the process has a Terminate Method. When the process does not have a Terminate Method, `process-terminate` can cause the other to exit in the middle of some sensitive piece of code.

See also

- `process-run-function`
- `current-process-set-terminate-method`
- `current-process-kill`

### process-unlock

**Function**

**Summary**

Unlocks a lock held by the current process.

**Package**

`mp`

**Signature**

`process-unlock lock &optional errorp => result`

**Arguments**

- `lock`: The lock.
- `errorp`: When this is `t`, an error is signaled if `*current-process*` is not the owner of `lock`. The default is `t`.

**Values**

- `result`: A boolean.

**Description**

Attempts to unlock `lock`. If `lock` is owned by `*current-process*`, then `process-unlock` decrements an internal count. If this count is then zero, `lock` is unlocked. Note that `process-unlock` relates only on Lisp processes.

`result` is `t` if the lock was released, and `nil` otherwise.

**Notes**

`process-sharing-unlock` is guaranteed to successfully unlock `lock`, but is not guaranteed to return, as described in “Guarantees and limitations when locking and unlocking” on page 278.
See also

lock
make-lock
process-exclusive-unlock
process-lock
with-lock
“Locks” on page 277

process-unstop

Function

Summary
Unstops a process.

Package
mp

Signature
process-unstop process => result

Arguments
process A mp:process object.

Values
result A boolean.

Description
The function process-unstop unstops the process process if it is stopped.

process must be a full process (that is, not one created by *current-process*).

If process was stopped (by process-stop), it is unstopped and resumes execution.

result is t if process was stopped, and nil otherwise.

There is a discussion of a typical use of process-unstop in the section “Stopping and unstopping processes” on page 296.

See also

process-stop
process-stopped-p
### process-wait

**Function**

**Summary**
Suspends the current process until a condition is true.

**Package**
mp

**Signature**

```lisp
process-wait wait-reason wait-function &rest wait-arguments =>
```

**Arguments**

- **wait-reason**
  A string describing the reason that the process is waiting.

- **wait-function**
  A function designator.

- **wait-arguments**
  The arguments that `wait-function` is applied to.

**Values**
None.

**Description**
The function `process-wait` suspends the current Lisp process until the predicate `wait-function` applied to `wait-arguments` returns true. This is tested periodically by the scheduler, but in situations where you want more control over the timing you should consider using `process-wait-local` instead of `process-wait` and then call `process-poke` in the process that is expected to make the `wait-function` return true.

`wait-function` has several limitations: it must not do a non-local exit, it should not have side effects and (since it is called frequently) it should be efficient.

Also, `wait-function` is called with interrupts blocked. It should therefore not allow interrupts, because this could cause deadlocks.

`wait-reason` allows you to find out why a process is waiting via the function `process-whostate`.

**See also**

- `process-poke`
- `process-wait-local`
process-wait-with-timeout
process-whostate
“Process Waiting and communication between processes” on page 280

**process-wait-for-event**

*Function*

**Summary**
Waits for an event in a “windowing friendly” way.

**Package**
mp

**Signature**

process-wait-for-event &key wait-reason wait-function
process-other-messages-p no-hang-p stop-at-user-operation-p => event

**Arguments**

- **wait-reason**
  A string or nil.

- **wait-function**
  A function designator.

- **process-other-messages-p**
  A generalized boolean.

- **no-hang-p**
  A generalized boolean.

- **stop-at-user-operation-p**
  A generalized boolean.

**Values**

- **result**
  An event or nil.

**Description**
The function `process-wait-for-event` calls `mailbox-wait-for-event` on the mailbox of the current process, after ensuring that the current process has a mailbox.

The arguments and value are interpreted as for `mailbox-wait-for-event`.

**See also**

`mailbox-wait-for-event`
**process-wait-function**  

*Function*

**Summary**
Returns a function that determines whether a process should continue to wait.

**Package**
mp

**Signature**
process-wait-function process => wait-function

**Arguments**
process  
A process.

**Values**
wait-function  
A function designator.

**Description**
The function process-wait-function returns the function that determines whether the Lisp process waits. The system periodically calls wait-function to decide whether to wake the process up.

wait-function is applied to wait-arguments, where both wait-function and wait-arguments were passed to process-wait.

**See also**
process-wait

**process-wait-local**  

*Function*

**Summary**
Has the same semantics as process-wait, but does not interact with the scheduler.

**Package**
mp

**Signature**
process-wait-local wait-reason function &rest args => t

**Arguments**
wait-reason  
A string.

function  
A function designator.

args  
Arguments passed to function.
The function `process-wait-local` suspends the current Lisp process until the predicate `function` applied to `args` returns true.

`process-wait-local` has same semantics as `process-wait`, but is "local", which here means that it does not interact with the scheduler. The scheduler does not call the wait function and hence never wakes the waiting process.

The wait function `function` is called only by the calling process, before going to sleep, and whenever it is "poked". A process is typically "poked" by calling `process-poke`, but all the other process managing functions (`process-unstop`, `process-interrupt`, `process-terminate`) also "poke" the process. Returning from any of the generic Process Waiting functions (see “Generic Process Wait functions” on page 281) or `cl:sleep` also implicitly pokes the process. A process may be also poked internally.

Because the wait function is checked only when the process is poked, it is the responsibility of the application to poke the process when it should check the wait function. This is the disadvantage of `process-wait-local` and `process-wait-local-with-timeout`.

**Note:** See `process-wait-local-with-periodic-checks` and `process-wait-local-with-timeout-and-periodic-checks` for functions that periodically check the wait functions.

One advantage of using the "local" waiters is that the wait function is called only by the waiting process. This means that the wait function does not have any of the restrictions that the wait function of `process-wait` has. In particular:

1. It does not matter if the wait function is not very fast. Note however, that it may be called several times, and not always in a predictable way, so it is better not to make it too slow or allocate much. You also cannot rely on any
side effect that is cumulative inside the wait function, except in the call that returns true (because this happens at most once).

2. If there is an unhandled error in the wait function it enters the debugger like normal Lisp code, so it is easier to debug.

3. The wait function is in the dynamic scope of the calling process, and so it sees all the dynamic bindings and can throw to all the catchers. That also means that all the handlers and restarts of the calling process are applicable in the wait function.

4. The wait function can itself call Process Waiting functions or `cl:sleep`, with a small caveat: since these functions may implicitly "poke" the process, if the wait function calls any of them and then returns false, it may be immediately called again (if it returns true then `process-wait-local` itself returns). Normally this is not a problem, because it is still waiting, but it does mean that the wait function is called more times than expected.

5. The wait function, because it can call Process Waiting functions, can use locks without causing errors. Note, however, that if the lock is held, it will cause an internal call to a Process Waiting function, which will "poke" the process and hence cause another call of the wait function (unless it returns true).

6. The wait function is visible in the output of the profiler. Another advantage of the "local" functions is that they do not interact with the scheduler and so they reduce the overhead of the scheduler.

`process-wait-local` always returns `t`.

See also

`process-poke`

`process-wait-local-with-periodic-checks`

`process-wait-local-with-timeout`
“Process Waiting and communication between processes” on page 280

**process-wait-local-with-periodic-checks**

*Function*

**Summary**
Like *process-wait-local*, but also calls the wait function periodically.

**Package**
mp

**Signature**

```
process-wait-local-with-periodic-checks wait-reason period function &rest args
```

**Arguments**

- *wait-reason* A string.
- *period* A positive real number.
- *function* A function designator.
- *args* Arguments passed to *function*.

**Description**

The function *process-wait-local-with-periodic-checks* is like *process-wait-local*, but also calls the wait function periodically.

The *period* is in seconds.

After each call to the function *wait-function*, the process sleeps at most *period* seconds, and then checks the wait function. If the process is poked while sleeping, it wakes up, checks the wait function, and then (if the wait function returns nil), sleeps again for at most *period* seconds.

**Notes**

The resolution of the period is dependent on the underlying operating system. Many systems give time-slices of few milliseconds, so the actual period may be out by a few milliseconds. In general, periods of 0.1 seconds or more are reasonably reliable, though not exact. Shorter periods become less and less reliable.
If the period is short, the wait function is called frequently, and hence there is more overhead for the system. With a reasonable wait function and a period of 0.1 or more, this overhead is probably insignificant. If you use shorter periods, or an expensive wait function, you may want to check what the overhead is. The easiest way to check is to make sure your system is such that the wait function returns `nil`, then run

```lisp
(ignore-errors ; just in case
  (sys:with-other-threads-disabled
   (time (mp:process-wait-local-with-timeout-and-
          periodic-checks
          "Timing" 5 period function args))))
```

When this form returns, compare the user and system times (which is what it actually used) to the elapsed time (which should be approximately 5 seconds). That will tell you what fraction of a "CPU" is used by the call. If the user and system time are less than 0.01 seconds, you may want to increase the time to get a more accurate number.

Warning: inside the scope of `with-other-threads-disabled`, the rest of the threads are disabled. So if your wait function ends up waiting for something that has to happen on another thread, your system will be deadlocked.

See also

- `process-poke`
- `process-wait-local`
- `process-wait-local-with-timeout-and-periodic-checks`

“Process Waiting and communication between processes” on page 280

**process-wait-local-with-timeout**  
*Function*

**Summary**  
Has the same semantics as `process-wait-with-timeout`, but does not interact with the scheduler.
Package  mp

Signature  process-wait-local-with-timeout wait-reason timeout function &rest args => result

Arguments  wait-reason  A string.
timeout    A non-negative real or nil.
function   A function designator.
args       Arguments passed to function.

Values     result  A boolean.

Description The function process-wait-local-with-timeout has same semantics as process-wait-with-timeout, but is "local", which here means that it does not interact with the scheduler. The scheduler does not call the wait function and hence never wakes the waiting process.

The timeout is in seconds.

The circumstances in which the function wait-function is called, and the restrictions on it, are as documented for process-wait-local except that the wait function can additionally be called when it times out.

process-wait-local-with-timeout returns t if a call to the wait function returns true. It returns nil if it times out.

See also  process-poke
          process-wait-local
          “Process Waiting and communication between processes” on page 280

process-wait-local-with-timeout-and-periodic-checks Function

Summary  Like process-wait-local-with-timeout, but also calls the wait function periodically.
The function `process-wait-local-with-timeout-and-periodic-checks` is like `process-wait-local-with-timeout`, but also calls the wait function periodically. The `timeout` and `period` are both in seconds. For information about the periodic calls, see `process-wait-local-with-periodic-checks`.

See also
- `process-poke`
- `process-wait-local-with-periodic-checks`
- `process-wait-local-with-timeout`
- “Process Waiting and communication between processes” on page 280

**process-wait-with-timeout**

**Function**

**Summary**
Suspend the current process until certain conditions are true, or until a timeout expires.

**Package**
`mp`

**Signature**
```
process-wait-with-timeout wait-reason timeout &optional
  wait-function &rest wait-arguments => bool
```
Arguments

- **wait-reason**: A string describing the reason that the process is waiting.
- **timeout**: A non-negative real or nil.
- **wait-function**: A function to test.
- **wait-arguments**: The arguments to apply to wait-function.

Values

- **bool**: A boolean.

Description

This function uses `process-wait` to suspend the current Lisp process until the predicate `wait-function` applied to `wait-arguments` returns true, or until `timeout` seconds have passed.

`wait-function` is called periodically by the scheduler, but in situations where you want more control over the timing you should consider using `process-wait-local` instead of `process-wait` and then call `process-poke` in the process that is expected to make the `wait-function` return true.

`wait-function` is called with interrupts blocked. It should therefore not allow interrupts, because this could cause deadlocks.

`bool` is nil if the timeout occurred before `wait-function` returned true. `bool` is true otherwise.

See also

- `process-join`
- `process-poke`
- `process-wait`
- `process-wait-local-with-timeout`
- `process-wait-local-with-timeout-and-periodic-checks`
- “Process Waiting and communication between processes” on page 280
**process-whostate**  

*Function*

**Summary**  
Returns the state of a process.

**Package**  
mp

**Signature**  
process-whostate process => reason

**Arguments**  
process  
A process.

**Values**  
reason  
A string.

**Description**  
The function `process-whostate` returns a string describing the state of the process. Depending on the state of `process`, `reason` can be:

- "Dead"
- "Stopped"
- "Sleeping"
- "Running"
- "Running (preempted)"

`reason` can also be the `wait-reason` of the process, as passed to `wait-processing-events`, `process-wait`, `mailbox-read` and so on.

`reason` can also be a string containing the `run-reasons`, as set by `(setf process-run-reasons)`.

**See also**  
wait-processing-events  
process-wait  
process-run-reasons

---

**processes-count**  

*Function*

**Summary**  
Returns the number of Lisp processes that are currently alive.
Package \textit{mp} \\
Signature \texttt{processes-count} => \textit{count} \\
Values \textit{count} \quad \text{A non-negative integer.} \\
Description The function \texttt{processes-count} returns the number of Lisp processes that are currently alive. 

The \textit{count} includes all processes that are alive, that is started executing and did not die. It does not include any thread that was started by foreign code, unless it calls into Lisp, in which case Lisp automatically generates a matching Lisp process which is included in the count.

In general processes can start and die so the real count may change by the time the function has returned. The only guarantee is that the count was accurate at some point between the time \texttt{processes-count} was called and the time it returns.

See also \texttt{list-all-processes}

\textbf{pushnew-to-process-private-property} \quad \textit{Function}

Summary Pushes a new value to a private property of the current process.

Package \textit{mp} \\
Signature \texttt{pushnew-to-process-private-property indicator value &key test => result} \\
Arguments \textit{indicator} \quad \text{A Lisp object.} \\
\textit{value} \quad \text{A Lisp object.} \\
\textit{test} \quad \text{A function designator for a function of two arguments.}
The MP Package

Values

result A list.

Description

The function pushnew-to-process-private-property pushes value to the value of the private property associated with indicator for the current process.

It behaves just like pushnew-to-process-property.

See also

process-private-property
pushnew-to-process-property
remove-process-private-property
get-process-private-property

pushnew-to-process-property

Function

Summary

Pushes a new value to a general property of a process.

Package

mp

Signature

pushnew-to-process-property indicator value &key process test => result

Arguments

indicator A Lisp object.
value A Lisp object.
process A process, or nil.
test A function designator for a function of two arguments.

Values

result A list.

Description

The function pushnew-to-process-property pushes value to the value of the property associated with indicator for the process process. It uses the function test to compare existing property values of process with value and does not push if one matches, in the same way as cl:pushnew.
The default value of test is '#eql.

If there is a property associated with indicator, the value of the property must be a list.

If process is not supplied or is nil, the current process (that is, the result of calling get-current-process) is used.

result is the new value of the process property.

The modification is done in a thread-safe way.

Notes
In the typical case when only the current process sets the property (even if other processes read it), private properties can be used, and are much faster in SMP LispWorks, because they do not need to deal with parallel setting. See process-private-property.

See also
process-property
process-private-property
remove-process-property

ps

Summary
Prints the processes in the system

Package
mp

Signature
ps =>

Arguments
None.

Values
None.

Description
Prints a list of the processes in the system, ordered by priority. (This function is analogous to the UNIX command ps.)
**remove-from-process-private-property**  
*Function*

Summary  
Removes a value from a private property of the current process.

Package  
*mp*

Signature  
`remove-from-process-private-property indicator value &key test => result`

Arguments  
- `indicator`  
  A Lisp object.
- `value`  
  A Lisp object.

Values  
- `result`  
  A list.

Description  
The function `remove-from-process-private-property` removes `value` from the value of the private property associated with `indicator` for the current process.

It behaves just like `remove-from-process-property`.

See also  
- `process-private-property`
- `remove-from-process-property`
- `remove-process-private-property`
- `get-process-private-property`

**remove-from-process-property**  
*Function*

Summary  
Removes a value from a general property of a process.

Package  
*mp*

Signature  
`remove-from-process-property indicator value &key process test => result`

Arguments  
- `indicator`  
  A Lisp object.
- `value`  
  A Lisp object.
A process, or \texttt{nil}.

A function designator for a function of two arguments.

A list.

The function \texttt{remove-from-process-property} removes \texttt{value} from the value of the property associated with \texttt{indicator} for the process \texttt{process}. It uses the function \texttt{test} to compare \texttt{value} with existing values, in the same way as \texttt{cl:remove}.

The default value of \texttt{test} is \texttt{#'eql}.

If there is a property associated with \texttt{indicator}, the value of the property must be a list.

If \texttt{process} is not supplied or is \texttt{nil}, the current process (that is, the result of calling \texttt{get-current-process}) is used.

\texttt{result} is the new value of the process property.

The modification is done in a thread-safe way.

In the typical case when only the current process sets the property (even if other processes read it), private properties can be used, and are much faster in SMP LispWorks, because they do not need to deal with parallel setting. See \texttt{process-private-property}.

\texttt{process-property} \newline \texttt{process-private-property} \newline \texttt{remove-process-property}

\texttt{Function}

Removes a property from the private properties of the current process.
The function `remove-process-private-property` removes the property associated with `indicator` from the private properties of the current process.

Note that removing a property is different from setting its value to `nil`, because when `process-private-property` is called with a `default` for a property that was removed, it returns the `default`, but for a property that was set to `nil` it returns `nil`.

See also `process-private-property`, `pushnew-to-process-private-property`, `remove-from-process-private-property`, `get-process-private-property`
Description

The function `remove-process-property` removes the general property associated with `indicator` from the process `process`.

If `process` is not supplied or is `nil`, the current process (that is, the result of calling `get-current-process`) is used.

Note that removing a property is different from setting its value to `nil`, because when `process-property` is called with a default for a property that was removed, it returns the default, but for a property that was set to `nil` it returns `nil`. `removedp` is true if the property was removed.

Notes

In the typical case when only the current process sets the property (even if other processes read it), private properties can be used, and are much faster in SMP LispWorks, because they do not need to deal with parallel setting. See `process-private-property`.

See also

`pushnew-to-process-property`
`remove-from-process-property`
`process-property`
`process-private-property`

---

**schedule-timer**

Function

Summary

Schedules a timer to expire at a given time after the start of the program.

Package

`mp`

Signature

```
schedule-timer timer absolute-expiration-time &optional repeat-time => timer
```

Arguments

- `timer` A timer.
The function `schedule-timer` schedules a timer to expire at a given time after the start of the program. The `timer` argument is a timer, returned by `make-timer` or `make-named-timer`. The `absolute-expiration-time` argument is a non-negative real number of seconds since the start of the program at which the timer is to expire. If `repeat-time` is specified, it is a non-negative real number of seconds that specifies a repeat interval. Each time the timer expires, it is rescheduled to expire after this repeat interval.

If the timer is already scheduled to expire at the time this function is called, it is rescheduled to expire at the time specified by the `absolute-expiration-time` argument. If that argument is `nil`, the timer is not rescheduled, but the repeat interval is set to the interval specified by the `repeat-time` argument.

The function `schedule-timer-relative` schedules a timer to expire at a time relative to the call to that function.

Example

The following example schedules a timer to expire 15 minutes after the start of the program and every 5 minutes thereafter.

```lisp
(setq timer
  (mp:make-timer 'print 10 *standard-output*))

#<Time Event : PRINT>  
(mp:schedule-timer timer 900 300)

#<Time Event : PRINT>
```
See also make-named-timer
make-timer
schedule-timer-milliseconds
schedule-timer-relative
schedule-timer-relative-milliseconds
timer-expired-p
timer-name
unschedule-timer
“Timers” on page 293

schedule-timer-milliseconds  Function

Summary  Schedules a timer to expire after a given amount of time.

Package  mp

Signature  schedule-timer-milliseconds timer absolute-expiration-time &optional repeat-time => timer

Arguments  timer  A timer.
absolute-expiration-time  A non-negative real number.
repeat-time  A non-negative real number.

Values  timer  A timer.

Description  The function schedule-timer-milliseconds schedules a timer to expire at a given time after the start of the program. The timer argument is a timer returned by make-timer or make-named-timer. The absolute-expiration-time argument is a non-negative real number of milliseconds since the start of the program at which the timer is to expire. If repeat-time is specified, it is a non-negative real number of milliseconds that specifies a repeat interval. Each time the timer expires, it is rescheduled to expire after this repeat interval.
If the timer is already scheduled to expire at the time this function is called, it is rescheduled to expire at the time specified by the `absolute-expiration-time` argument. If that argument is `nil`, the timer is not rescheduled, but the repeat interval is set to the interval specified by the `repeat-time` argument.

The function `schedule-timer-relative-milliseconds` schedules a timer to expire at a time relative to the call to that function.

**Notes**

`schedule-timer-milliseconds` has the same precision as `schedule-timer`, but may avoid some allocation when computing the time.

**Example**

The following example schedules a timer to expire 15 minutes after the start of the program and every 5 minutes thereafter.

```lisp
(setq timer
  (mp:make-timer 'print 10 *standard-output*))
#<Time Event : PRINT>
(mp:schedule-timer-milliseconds timer 900000 300000)
#<Time Event : PRINT>
```

**See also**

`make-named-timer`

`make-timer`

`schedule-timer`

`schedule-timer-relative`

`schedule-timer-relative-milliseconds`

`timer-expired-p`

`timer-name`

`unschedule-timer`
**schedule-timer-relative**

*Function*

**Summary**
Schedules a timer to expire at a given time after this function is called.

**Package**
`mp`

**Signature**

```
schedule-timer-relative timer relative-expiration-time
&optional repeat-time => timer
```

**Arguments**

- `timer` A timer
- `relative-expiration-time` A non-negative real
- `repeat-time` A non-negative real

**Values**

- `timer` A timer

**Description**

The function `schedule-timer-relative` schedules a timer to expire at a given time after the call to the function. The `timer` argument is a timer returned by `make-timer` or `make-named-timer`. The `relative-expiration-time` argument is a non-negative real number of seconds after the call to the function at which the timer is to expire. If `repeat-time` is specified, it is a non-negative real number of seconds that specifies a repeat interval. Each time the timer expires, it is rescheduled to expire after this repeat interval.

If the timer is already scheduled to expire at the time this function is called, it is rescheduled to expire at the time specified by the `relative-expiration-time` argument. If that argument is `nil`, the timer is not rescheduled, but the repeat interval is set to the interval specified by the `repeat-time` argument.

The function `schedule-timer` schedules a timer to expire at a time relative to the start of the program.
Example

The following example schedules a timer to expire 5 seconds after the call to `schedule-timer-relative` and every 5 seconds thereafter.

```
(setq timer
  (mp:make-timer 'print 10 *standard-output*))
#<Time Event : PRINT>
(mp:schedule-timer-relative timer 5 5)
#<Time Event : PRINT>
```

See also

`make-named-timer`
`make-timer`
`schedule-timer`
`schedule-timer-milliseconds`
`schedule-timer-relative-milliseconds`
`timer-expired-p`
`timer-name`
`unschedule-timer`

**schedule-timer-relative-milliseconds**

*Function*

**Summary**

Schedules a timer to expire at a given time after this function is called.

**Package**

`mp`

**Signature**

`schedule-timer-relative-milliseconds timer relative-expiration-time &optional repeat-time => timer`

**Arguments**

- `timer` A timer.
- `relative-expiration-time` A non-negative real.
- `repeat-time` A non-negative real.

**Values**

- `timer` A timer.
The function `schedule-timer-relative-milliseconds` schedules a timer to expire at a given time after the call to the function. The timer argument is a timer returned by `make-timer` or `make-named-timer`. The relative-expiration-time argument is a non-negative real number of milliseconds after the call to the function at which the timer is to expire. If repeat-time is specified, it is a non-negative real number of milliseconds that specifies a repeat interval. Each time the timer expires, it is rescheduled to expire after this repeat interval.

If the timer is already scheduled to expire at the time this function is called, it is rescheduled to expire at the time specified by the relative-expiration-time argument. If that argument is `nil`, the timer is not rescheduled, but the repeat interval is set to the interval specified by the repeat-time argument.

The function `schedule-timer-milliseconds` schedules a timer to expire at a time relative to the start of the program.

`schedule-timer-relative-milliseconds` has the same precision as `schedule-timer-relative`, but may avoid some allocation when computing the time.

The following example schedules a timer to expire 5 seconds after the call to `schedule-timer-relative-milliseconds` and every 5 seconds thereafter.

```
(setq timer
  (mp:make-timer 'print 10 *standard-output*))
#<Time Event : PRINT>
(mp:schedule-timer-relative-milliseconds timer 5000 5000)
#<Time Event : PRINT>
```

See also

- `make-named-timer`
- `make-timer`
- `schedule-timer`
semaphore

Summary
A class of objects for synchronizing access to a shared resource that contains some number of available units.

Package
mp

Superclasses
t

Description
Instances of the system class semaphore are used for synchronizing access to a shared resource that contains some number of available units. They are made by make-semaphore and are used semaphore-acquire and semaphore-release.

See also
make-semaphore
semaphore-acquire
semaphore-release
“Counting semaphores” on page 287

semaphore-acquire

Summary
Acquires units from a semaphore.

Package
mp

Signature
semaphore-acquire sem &key timeout wait-reason count => flag

Arguments
sem A semaphore.
timeout A non-negative real or nil.
wait-reason A string or nil.
count A non-negative fixnum.

Values flag A generalized boolean.

Description The function `semaphore-acquire` acquires `count` units from the `semaphore` `sem`.

It attempts to atomically decrement the semaphore's unit count by `count`. If this gives a non negative result, then it changes the semaphore’s unit count accordingly and returns true. The default value of `count` is 1.

However, if decrementing the semaphore’s unit count would result in a negative number then `semaphore-acquire` waits until the semaphore’s unit count is larger than `count` and tries again. If `wait-reason` is true, then it is used as the thread’s `wait-reason` when waiting for the semaphore.

If `timeout` is `nil`, `semaphore-acquire` can wait forever. Otherwise, if the semaphore count cannot be decremented within `timeout` seconds, then `semaphore-acquire` returns false and the semaphore is unaffected. Pass `timeout 0` if you do not want to wait at all.

Notes You can get the current unit count of a semaphore by calling `semaphore-count`.

See also `semaphore make-semaphore semaphore-count semaphore-release semaphore-wait-count` "Counting semaphores" on page 287
**semaphore-count**

*Function*

Summary  Gets the current unit count of a *semaphore*.

Package  mp

Signature  

Arguments  

Values  

Description  The function *semaphore-count* returns the current unit count of the *semaphore* `sem`. The value is 0 if the semaphore has no unit remaining.

Notes  The current unit count value can change in the semaphore after calling *semaphore-count*.  
The value returned by *semaphore-count* is never negative.

See also  

`semaphore`

`make-semaphore`

`semaphore-acquire`

`semaphore-release`

`semaphore-wait-count`

“Counting semaphores” on page 287

**semaphore-name**

*Function*

Summary  Gets the name of a *semaphore*.

Package  mp

Signature  

Arguments  

Values  


Values

name

An object.

Description

The function `semaphore-name` returns the name that `semaphore` `sem` was given when it was created.

See also

`semaphore`

`make-semaphore`

“Counting semaphores” on page 287

semaphore-release

Function

Summary

Releases units back to a `semaphore`.

Package

`mp`

Signature

`semaphore-release sem &key count => flag`

Arguments

`sem` A `semaphore`.

`count` A non negative fixnum.

Values

`flag` A generalized boolean.

Description

The function `semaphore-release` releases count units back to the `semaphore` `sem`.

It atomically increments the semaphore’s unit count by `count` (which defaults to 1).

The returned `flag` is true if any other thread was waiting for the semaphore and false otherwise.

See also

`semaphore`

`make-semaphore`

`semaphore-acquire`

`semaphore-count`

`semaphore-wait-count`

“Counting semaphores” on page 287
semaphore-wait-count  

Function

Summary  Get the current wait count of a semaphore.

Package  mp

Signature  semaphore-wait-count sem => wait-count

Arguments  sem  A semaphore.

Values  wait-count  A non negative fixnum.

Description  The function semaphore-wait-count returns the current number of units that other threads are waiting for from the semaphore sem. The value wait-count is 0 if the semaphore has no thread waiting for it.

Notes  The value can change in the semaphore after calling semaphore-wait-count.

See also  semaphore  
make-semaphore  
semaphore-acquire  
semaphore-count  
semaphore-release  
“Counting semaphores” on page 287

set-funcall-async-limit  

Function

Summary  Limit the number of parallel asynchronous calls.

Package  mp

Signature  set-funcall-async-limit new-limit => result
Arguments new-limit
An integer in the exclusive range (0,100000) or nil.

Values result
An integer in the exclusive range (0,100000).

Description
The function set-funcall-async-limit limits the number of asynchronous calls (by funcall-async or funcall-async-list) which can run in parallel. Further asynchronous calls are queued, and when a running call finishes another call starts.

When new-limit is an integer the limit is set to new-limit, and result is the previous limit.

When new-limit is nil, the limit is not changed and result is the current limit.

The default limit is 5, which is adequate if funcall-async and/or funcall-async-list are only used occasionally. If you use them often, you may want to increase this limit to between 10 and 30. A larger limit probably does not make sense.

See also
funcall-async
funcall-async-list

### simple-lock-and-condition-variable-wait

**Function**

**Summary**
A variant of lock-and-condition-variable-wait with a simpler lambda list.

**Package**
mp

**Signature**
simple-lock-and-condition-variable-wait lock lock-timeout condvar condvar-timeout predicate &rest args => result

**Arguments**
lock A lock.
lock-timeout A non-negative real or nil.
**condvar**  
A condition-variable.

**condvar-timeout**  
A non-negative real or nil.

**predicate**  
A function designator.

**args**  
Arguments to **predicate**.

**Values**  
**result**  
See below.

**Description**  
The function **simple-lock-and-condition-variable-wait** is a variant of **lock-and-condition-variable-wait** that does not take keyword arguments. Also it takes the arguments of the **predicate** as &rest. It interprets and acts on the arguments just like **lock-and-condition-variable-wait**.

**simple-lock-and-condition-variable-wait** returns the result of calling **predicate** or the wait, like **lock-and-condition-variable-wait** when **return-function** is not supplied.

**Notes**  
**simple-lock-and-condition-variable-wait** does not take wait reason arguments, so you should give names to the **lock** and the **condition-variable** **condvar** for debugging (by passing **name** in **make-lock** and **make-condition-variable**).

**See also**  
**condition-variable-wait**  
**lock-and-condition-variable-wait**  
**lock-and-condition-variable-signal**  
**lock-and-condition-variable-broadcast**  
**condition-variable-signal**  
**condition-variable-broadcast**  
“Condition variables” on page 285  
“Locks” on page 277
### symeval-in-process

**Function**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Reads the value of symbol which is dynamically bound in a given process.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>mp</td>
</tr>
<tr>
<td>Signature</td>
<td><code>symeval-in-process symbol process =&gt; value, flag</code></td>
</tr>
<tr>
<td></td>
<td><code>(setf symeval-in-process) value symbol process =&gt; value</code></td>
</tr>
<tr>
<td>Arguments</td>
<td>symbol A symbol</td>
</tr>
<tr>
<td></td>
<td>process A process</td>
</tr>
<tr>
<td>Values</td>
<td>value A Lisp object</td>
</tr>
<tr>
<td></td>
<td>flag One of <code>t</code>, <code>nil</code> or the keyword <code>:unbound</code></td>
</tr>
<tr>
<td>Description</td>
<td>The function <code>symeval-in-process</code> reads the value of the symbol <code>symbol</code> in the process <code>process</code> if it is bound dynamically. The global value of <code>symbol</code> is never returned.</td>
</tr>
<tr>
<td></td>
<td>If <code>symbol</code> is not bound in <code>process</code>, then <code>value</code> and <code>flag</code> are both <code>nil</code>. If <code>symbol</code> is bound in <code>process</code> but <code>makunbound</code> has been called within the dynamic scope of the binding, <code>value</code> is <code>nil</code> and <code>flag</code> is <code>:unbound</code>. Otherwise, <code>value</code> is the value of <code>symbol</code> and <code>flag</code> is <code>t</code>.</td>
</tr>
<tr>
<td></td>
<td>In addition, the form</td>
</tr>
<tr>
<td></td>
<td><code>(setf (symeval-in-process symbol process) value)</code></td>
</tr>
<tr>
<td></td>
<td>sets the value of <code>symbol</code> to <code>value</code> in <code>process</code>. It is an error if <code>process</code> has no binding for <code>symbol</code>. This <code>setf</code> form returns <code>value</code> as specified by Common Lisp.</td>
</tr>
<tr>
<td>Notes</td>
<td><code>symeval-in-process</code> is mostly intended for debugging. It is OK to call it on a thread known to be idle, or in <code>process-wait</code> or <code>process-stop</code>, but it should not be called while the thread is running.</td>
</tr>
</tbody>
</table>

1503
### timer-expired-p

**Function**

**Summary**
Returns `t` if a given timer has expired or is about to expire.

**Package**
`mp`

**Signature**
`timer-expired-p timer &optional delta => bool`

**Arguments**
- **timer**: A timer
- **delta**: A non-negative real

**Values**
- **bool**: A boolean

**Description**
The function `timer-expired-p` returns `t` if the specified timer is not scheduled to expire or is scheduled to expire within the number of seconds specified by the `delta` argument after the call to `timer-expired-p`. Otherwise, the function returns `nil`.

The `timer` argument is a timer, returned by `make-timer` or `make-named-timer`. The `delta` argument, if supplied, is a non-negative real number of seconds.

**Example**
```
(setq timer
  (mp:make-timer 'print 10 *standard-output*))
#<Time Event : PRINT>
(mp:schedule-timer-relative timer 5)
#<Time Event : PRINT>
(mp:timer-expired-p timer)
NIL
```

**See also**
- `make-named-timer`
- `make-timer`
- `schedule-timer`
- `schedule-timer-milliseconds`
**timer-name**  

**Function**

**Summary**  
Returns the name of a specified timer.

**Package**  
mp

**Signature**  
`timer-name timer => name`

**Signature**  
`(setf timer-name) name timer => name`

**Arguments**  
*timer*  
A timer

**Values**  
*name*  
A string

**Description**  
The function **timer-name** returns the name of the specified *timer*. The *timer* argument is a timer returned by **make-timer** or **make-named-timer**. If the timer has no name, **timer-name** returns **nil**.

The name of a timer created by either **make-timer** or **make-named-timer** can be set by means of the following syntax:

```
(setf (mp:timer-name timer) name)
```

**Example**
```
(setq timer
    (mp:make-timer 'print 10 *standard-output*))

(defvar *timers* (list timer))

;; Time Event : PRINT>

(mp:timer-name timer)
NIL

(setf (mp:timer-name timer) 'timer-1)

(TIMER-1)
```
(mp:timer-name timer)
TIMER-1

See also
make-named-timer
make-timer
schedule-timer
schedule-timer-milliseconds
schedule-timer-relative
timer-expired-p
unschedule-timer

unnotice-fd

Function

Summary
Removes a file descriptor from the set of interesting input file descriptors.

Package
mp

Signature
unnotice-fd fd

Arguments
fd A file descriptor

Values
None.

Description
The function unnotice-fd removes fd from the set of fds that cause LispWorks to wake up when they contain input.
This function is not implemented on Microsoft Windows.

See also
notice-fd

unschedule-timer

Function

Summary
Unschedules a scheduled timer
unschedule-timer timer => result

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timer</td>
<td>A timer</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>A timer or nil</td>
</tr>
</tbody>
</table>

Description

If the specified timer has been scheduled to expire at a time after the call to unschedule-timer, this function unschedules the timer and returns the timer. Otherwise, the function returns nil.

The argument is a timer, returned by make-timer or make-named-timer.

Example

```lisp
(setq timer
  (mp:make-timer 'print 10 *standard-output*))
#<Time Event : PRINT>
(mp:schedule-timer-relative timer 60)
#<Time Event : PRINT>
(mp:unschedule-timer timer)
#<Time Event : PRINT>
(mp:timer-expired-p timer)
T
```

See also

make-named-timer
make-timer
schedule-timer
schedule-timer-milliseconds
schedule-timer-relative
timer-expired-p
timer-name
**wait-processing-events**  
*Function*

**Summary**  
Waits processing events.

**Package**  
mp

**Signature**  
\[ \text{wait-processing-events \ timeout \ \&key \ wait-reason \ wait-function \ wait-args =\> result} \]

**Arguments**
- \( \text{timeout} \): A non-negative real or nil.
- \( \text{wait-reason} \): A string.
- \( \text{wait-function} \): A function designator.
- \( \text{wait-args} \): A list.

**Values**
- \( \text{result} \): t or nil

**Description**  
The function \( \text{wait-processing-events} \) does not return until one of two conditions is met:

- \( \text{timeout} \) is non-nil and \( \text{timeout} \) seconds have passed.
  
  In this case, \( \text{result} \) is nil.

- \( \text{wait-function} \) returns a true value.
  
  In this case, \( \text{result} \) is t.

\( \text{wait-reason} \) provides the value returned by \( \text{process-whos-state} \) when called on the current process.

\( \text{wait-function} \) is called periodically with arguments \( \text{wait-args} \).
\( \text{wait-function} \) may be called many times and in several places. Therefore \( \text{wait-function} \) should be fast and make no assumptions about its dynamic context.

\( \text{wait-processing-events} \) processes all events sent to the current process, including system events such as window messages on Microsoft Windows, and objects sent by other processes via \( \text{process-send} \). In the latter case, the objects must be lists of the form \((\text{function} \ . \ \text{arguments})\), which cause \( \text{function} \) to be applied to \( \text{arguments} \) (the values are discarded).
wait-processing-events is a useful alternative to sleep in a situation where you want to process events to see window updates and so on.

See also  
process-send  
process-whostate  

with-exclusive-lock  
Macro

Summary  Holds a sharing lock in exclusive mode while evaluating its body, and then unlocks the lock.

Package  mp

Signature  with-exclusive-lock (sharing-lock &optional whostate timeout) &body body => results

Arguments  sharing-lock  A sharing lock.
whostate  The status of the process while sharing-lock is locked, as seen in the Process Browser.
timeout  A non-negative real or nil.
body  The forms to execute

Values  results  The values returned from evaluating body.

Description  The macro with-exclusive-lock is the same as with-lock, except that sharing-lock must be a "sharing" lock, that is, created with the argument sharing true in make-lock. It waits until sharing-lock is completely free, that is, not locked in a sharing mode and is not locked in exclusive mode by another thread. It then locks sharing-lock in exclusive mode, evaluates body and unlocks sharing-lock.

Notes  It is not possible to hold an exclusive lock in the scope of a sharing-lock on the same lock, and trying to do it will cause
the process to hang. Whether it is possible to hold an exclusive lock inside an exclusive-lock of the same lock is determined by the recursivep argument in make-lock.

See also

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-lock</td>
<td></td>
</tr>
<tr>
<td>with-lock</td>
<td></td>
</tr>
<tr>
<td>“Locks” on page 277</td>
<td></td>
</tr>
</tbody>
</table>

### with-interrupts-blocked

**Macro**

**Summary**
Evaluates code with interrupts blocked.

**Package**
mp

**Signature**

```lisp
with-interrupts-blocked &body body => results
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>body</td>
<td>Code</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>results</td>
<td>Values returned by evaluating body.</td>
</tr>
</tbody>
</table>

**Description**
Evaluates body with interrupts blocked. This actually expands to

```lisp
(mp:allowing-block-interrupts t ,@body)
```

which means it also allows you to change the blocking of interrupts.

See the entry for allowing-block-interrupts for full details.

See also

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allowing-block-interrupts</td>
<td></td>
</tr>
</tbody>
</table>

### with-lock

**Macro**

**Summary**
Executes a body of code while holding a lock.
with-lock

**Signature**

\[
\text{with-lock (lock &optional whostate timeout) &body body => result}
\]

**Arguments**

- **lock**: The lock.
- **whostate**: The status of the process while lock is locked, as seen in the Process Browser.
- **timeout**: A non-negative real or nil.
- **body**: The forms to execute.

**Values**

- **result**: The result of executing body.

**Description**

with-lock executes body while holding lock, and unlocks lock when body exits. This is the recommended way of using a lock. The value of body is returned normally. body is not executed if lock could not be locked, in which case, with-lock returns nil. The timeout and whostate are used as specified by process-lock.

**See also**

make-lock
process-lock
process-unlock
with-exclusive-lock
with-sharing-lock
“Locks” on page 277

---

**with-sharing-lock**

**Macro**

**Summary**

Holds a lock in shared mode while executing a body of code.

**Package**

mp

**Signature**

\[
\text{with-sharing-lock (sharing-lock &optional whostate timeout) &body body => results}
\]
Arguments

- sharing-lock: A sharing lock.
- whostate: The status of the process while sharing-lock is locked, as seen in the Process Browser.
- timeout: A non-negative real or nil.
- body: The forms to execute

Values

- results: The values returned from evaluating body.

Description

The macro `with-sharing-lock` is like `with-lock`, but sharing-lock must be a "sharing" lock and will be locked in shared mode. That means that other threads can also lock it in shared mode.

Before locking, `with-sharing-lock` waits for sharing-lock to be free of any exclusive lock, but it does not check for other shared mode use of the same lock.

Notes

It is possible to lock for sharing inside the scope of sharing lock and inside the scope of an exclusive lock.

See also

- `make-lock`
- `with-lock`
- "Locks" on page 277

**without-interrupts**

**Macro**

Summary

Causes any interrupts that occur during the execution of a body of code to be queued, in non-SMP LispWorks only.

Package

`mp`

Signature

`without-interrupts &rest body => result`

Arguments

- body: The forms to execute while interrupts are queued.
Values  

result  
The result of executing body.

Description  
While body is executing, all interrupts (for example, preemption, keyboard break etc.) are queued. They are executed when body exits.

Notes  
without-interrupts is not supported in SMP LispWorks, that is on Microsoft Windows, Mac OS X, Linux, FreeBSD, AIX and x86/x64 Solaris platforms.

Example  
To ensure that the seconds and milliseconds slots are always consistent in non-SMP LispWorks, you can use without-interrupts within the function which sets them.

(defstruct elapsed-time  
  seconds  
  milliseconds)

(defun update-elapsed-time-atomically  
  (elapsed-time seconds milliseconds)  
  (mp:without-interrupts  
    (setf (elapsed-time-seconds elapsed-time) seconds  
          (elapsed-time-milliseconds elapsed-time)  
          milliseconds)))

See also  
without-preemption

without-preemption  
Macro

Summary  
Identifies forms which should not be preempted during execution, in non-SMP LispWorks only

Package  
mp

Signature  
without-preemption &rest body => result

Arguments  
body  
The forms to be evaluated atomically.

Values  
result  
The result of executing body.
Description  Identifies forms which should not be preempted during execution.

Notes  \texttt{without-preemption} is not supported in SMP LispWorks, that is on Microsoft Windows, Mac OS X, Linux, FreeBSD, AIX and x86/x64 Solaris platforms.

\textbf{yield}  \hfill Function

Summary  Allows preemption to happen in low safety code.

Package  \texttt{mp}

Signature  \texttt{yield}

Arguments  None.

Values  None.

Description  Normally code compiled at safety 0 cannot be preempted because the necessary checks are omitted. This can be overcome by calling \texttt{yield} at regular intervals. Usually there is no need to call this if you use functions from the \texttt{common-lisp} package because these are not compiled at safety 0, but for example if you find that preemption is not working in a loop with no function calls, \texttt{yield} can be useful. Note that \texttt{process-allow-scheduling} also allows preemption, but also checks the wait functions of other processes.

See also  \texttt{process-allow-scheduling}
The PARSERGEN Package

This chapter describes symbols available in the PARSERGEN package, the Lisp-Works parser generator.

This functionality is discussed in detail in Chapter 21, “The Parser Generator”.

defparser

Macro

Summary
Creates a parsing function of the given name for the grammar defined.

Package
parsergen

Signature

defparser name-and-options {rule}* => parsing-function
name-and-options ::= name | (name [[option]])
option ::= :accept-without-eoi-p accept-without-eoi-p
rule ::= normal-rule | error-rule | combined-rule
normal-rule ::= ((non-terminal {grammar-symbol}*) {form}*)
error-rule ::= ((non-terminal :error) {form}*)
combined-rule ::= (non-terminal {combined-rule-clause}*)
combined-rule-clause ::= (combined-rule-lhs {form}*)
combined-rule-lhs ::= ({grammar-symbol}*) | (:error)

Arguments

name The name of the parser.
accept-without-eoi-p A generalized boolean.

The rules define the productions of the grammar and the associated forms define the semantic actions for the rules.

Values

parsing-function The symbol name of the parsing function.

Description

defparser creates a parsing function of the given name for the grammar defined. The parsing function is defined as if by:

(defun <name> (lexer &optional (symbol-to-string #'identify))

The lexer parameter is a function of no arguments that returns two values: the next grammar token on the input and the associated semantic value.

The optional symbol-to-string function can be used to define a printed representation of the grammar tokens. The function should take a grammar symbol as its single argument and returns an object to be used as a print representation for the grammar token.

When accept-without-eoi-p is true, the parser accepts the input as soon as the top level rule matches completely rather than waiting for end of input (eoi). The default value of accept-without-eoi-p is false.

For a full description and examples, see Chapter 21, “The Parser Generator”.


This chapter describes the symbols available in the SERIAL-PORT package.
The Serial Port functionality is loaded into LispWorks by evaluating

```
(require "serial-port")
```

See open-serial-port for platform-specific details.

**open-serial-port**

*Function*

**Summary**
Attempts to open the named serial port and return a serial-port object.

**Package**
serial-port

**Signature**

**Arguments**
name
A string naming a serial port.
The SERIAL-PORT Package

This chapter applies only to LispWorks for Windows

The function `open-serial-port` attempts to open the serial port `name` and return a `serial-port` object.

On Windows, `name` is passed directly to `CreateFile()`. For ports COMn where n > 9, you must take care to pass the real port name expected by Windows. At the time of writing this issue is documented at http://support.microsoft.com/kb/115831.

On non-Windows platforms, `name` should be the device file name (for example "/dev/cu.usbmodem14111").

If any of `baud-rate`, `data-bits`, `stop-bits` and `parity` are supplied then the corresponding serial port settings are changed. The values of `baud-rate` and `data-bits` should each be an appropriate integer. The value of `stop-bits` should be 1, 1.5 (Windows only) or 2. The value of `parity` should be one of the keywords :even, :none or :odd, or on Windows, :mark or :space.

The arguments `cts-flow-p` and `dsr-flow-p` control whether write operations respond to CTS and DSR flow control. A non-nil value means that the corresponding flow control is used. Note that `dsr-flow-p` is only supported on Windows.

The arguments `dtr` and `rts` control whether read operations generate DTR or RTS flow control. If the value is :handshake then the corresponding flow control signal is generated automatically. If the value is nil or t then the initial state of the flow control signal is set and automatic flow control is not used. See `set-serial-port-state` for manual flow control. Note: the value :handshake for `dtr` is only supported on Windows.

The argument `read-interval-timeout` can be used to control the maximum time to wait between each input character. The
value :none means that reading will not wait for characters at all, only returning whatever is already in the input buffer

The arguments read-total-base-timeout and read-total-byte-timeout can be used to control the maximum time to wait for a sequence of characters. The arguments write-total-base-timeout and write-total-byte-timeout can be used to control the maximum time to wait when transmitting a sequence of characters. For both reading and writing the timeout is given by the expression:

\[ \text{base_timeout} + \text{nchars} \times \text{byte_timeout} \]

The default value of each of read-total-base-timeout, read-total-byte-timeout, write-total-base-timeout and write-total-byte-timeout is \text{nil} and this means that the corresponding parameter in the OS is left unchanged and there is zero timeout. Otherwise the value should be a non-negative real number specifying a timeout in seconds.

See also

\begin{itemize}
  \item \text{close-serial-port}
  \item \text{set-serial-port-state}
\end{itemize}

\section*{close-serial-port}

\textbf{Function}

\begin{description}
  \item[Summary] Closes a serial port
  \item[Package] \text{serial-port}
  \item[Signature] close-serial-port \text{serial-port}
  \item[Arguments] \text{serial-port} \hspace{1em} A \text{serial-port} object.
  \item[Description] The function \text{close-serial-port} closes the serial port associated with the given \text{serial-port} object.

  If \text{serial-port} is already closed, an error is signaled.
\end{description}
get-serial-port-state

Function

Summary
Queries various aspects of the state of a serial port.

Package serial-port

Signature
get-serial-port-state serial-port keys => state

Arguments
serial-port A serial-port object.
keys A list of keywords.

Values state A list.

Description
The function get-serial-port-state queries various aspects of the state of the serial port associated with serial-port.

The argument keys should be a list of one or more of the keywords :dsr and :cts. These cause get-serial-port-state to check the DSR and CTS lines respectively.

The result state is a list giving the state of each line in the same order as they appear in the argument keys.

serial-port

Class

Summary The class of objects representing serial ports.

Package serial-port

Description The class serial-port is the class of objects representing serial ports. These are constructed by open-serial-port - do not create them directly.
See also open-serial-port

read-serial-port-char

Function

Summary
Reads a character from a serial port.

Package
serial-port

Signature
read-serial-port-char serial-port &optional timeout-error-p
timeout-char => char

Arguments
serial-port A serial-port object.
timeout-error-p A boolean.
timeout-char A character.

Values
char A character.

Description
The function read-serial-port-char reads and returns a character from the serial port associated with serial-port.

A timeout will occur if the character is not available before the read timeout (as specified by values given when the serial port was opened by open-serial-port).

When a timeout occurs, if timeout-error-p is non-nil, then an error of type serial-port-timeout is signaled, otherwise timeout-char is returned. The default value of timeout-error-p is t.

See also read-serial-port-string

read-serial-port-string

Function

Summary
Reads a string from a serial port.
The SERIAL-PORT Package

This chapter applies only to LispWorks for Windows

Package serial-port

Signature read-serial-port-string string serial-port &optional timeout-error-p &key start end => nread

Arguments

string A string.
serial-port A serial-port object.
timeout-error-p A boolean.
start, end Bounding index designators for string.

Values nread An integer.

Description The function read-serial-port-string reads characters from the serial port associated with serial-port and places them in string, bounded by start and end.

The default values of start and end are 0 and nil (interpreted as the length of string) respectively. The number of characters requested is the difference between end and start.

If the number of characters actually read, nread, is less than the number requested, then if timeout-error-p is non-nil an error of type serial-port-timeout is signaled.

If nread is the number of characters requested, or if timeout-error-p is nil, nread is returned.

The default value of timeout-error-p is t.

See also read-serial-port-char

serial-port-input-available-p Function

Summary Checks whether a character is available on a serial port.

Package serial-port
This chapter applies only to LispWorks for Windows

**Signature**

\[\text{serial-port-input-available-p \ serial-port } \Rightarrow \text{result}\]

**Arguments**

- **serial-port**
  A `serial-port` object.

**Values**

- **result**
  A boolean.

**Description**

The function `serial-port-input-available-p` checks the serial port associated with `serial-port` to see if a character is available. `result` is `t` if input is available, and `nil` otherwise.

---

**set-serial-port-state**

**Function**

**Summary**

Changes various aspects of the state of a serial port.

**Package**

`serial-port`

**Signature**

\[\text{set-serial-port-state \ serial-port \ &key \ dtr \ rts \ break}\]

**Arguments**

- **serial-port**
  A `serial-port` object.

- **dtr**
  A boolean.

- **rts**
  A boolean.

- **break**
  A boolean.

**Description**

The function `set-serial-port-state` changes various aspects of the state of the serial port associated with `serial-port`.

The argument `dtr`, if supplied, controls the DTR line. A true value means set and `nil` means clear. If `dtr` is not supplied, the state is unchanged.

The argument `rts` controls the RTS line in the same way.

The argument `break` controls the break state of the data line in the same way.
**wait-serial-port-state**

**Summary**
Waits for some aspect of the state of a serial port to change.

**Package**
serial-port

**Signature**
wait-serial-port-state serial-port keys &key timeout => result

**Arguments**
- *serial-port* A serial-port object.
- *keys* A list of keywords.
- *timeout* A number.

**Values**
result A list.

**Description**
The function `wait-serial-port-state` waits for some state in the serial port associated with `serial-port` to change.

The argument `keys` should be a list of one or more of the keywords :cts, :dsr, :err, :ring, :rlosd and :break.

`result` is a list giving the keys for which the state has changed.

If `timeout` is non-nil then the function will return `nil` after that many seconds even if the state has not changed.

**write-serial-port-char**

**Summary**
Writes a character to a serial port.

**Package**
serial-port

**Signature**
write-serial-port-char char serial-port &optional timeout-error-p => char

**Arguments**
- *char* A character.
- *serial-port* A serial-port object.
- *timeout-error-p* A boolean.
Values
char
A character.

Description
The function **write-serial-port-char** writes the character *char* to the serial port associated with *serial-port*, and returns *char*.

A timeout will occur if the character cannot be written before the write timeout (as specified by values given when the serial port was opened by **open-serial-port**).

When a timeout occurs, if *timeout-error-p* is non-nil, then an error of type **serial-port-timeout** is signaled, otherwise *nil* is returned. The default value of *timeout-error-p* is t.

See also

**write-serial-port-string**

*Function*

**Summary**
Writes a string to a serial port.

**Package**
serial-port

**Signature**

```lisp
write-serial-port-string string serial-port &optional
timeout-error-p &key start end => nwritten
```

**Arguments**

- **string**
  A string.
- **serial-port**
  A serial-port object.
- **timeout-error-p**
  A boolean.
- **start, end**
  Bounding index designators for *string*.

**Values**

- **result**
  The string *string* or *nil*.

**Description**
The function **write-serial-port-string** writes characters from the subsequence of *string* bounded by *start* and *end* to the serial port associated with *serial-port*. 
The default values of \texttt{start} and \texttt{end} are 0 and \texttt{nil} (interpreted as the length of \texttt{string}) respectively.

If the characters are successfully written then \texttt{string} is returned.

A timeout will occur if the characters cannot be written before the write timeout (as specified by values given when the serial port was opened by \texttt{open-serial-port}).

When a timeout occurs, if \texttt{timeout-error-p} is non-nil, then an error of type \texttt{serial-port-timeout} is signaled, otherwise \texttt{nil} is returned. The default value of \texttt{timeout-error-p} is \texttt{t}.

\textbf{See also} \texttt{write-serial-port-char}
This chapter describes the symbols available in the SQL package which implements Common SQL. You should use this chapter in conjunction with Chapter 23, “Common SQL”. In particular that chapter contains more information about the Oracle LOB interface (that is, those functions with names beginning sql:ora-lob-).

On Microsoft Windows, Linux, x86/x64 Solaris, FreeBSD, AIX and Mac OS X, Common SQL is included only in LispWorks Enterprise Edition.

accepts-n-syntax Function

Summary       Return whether a database connection accepts or requires the N syntax for non-ASCII strings.

Package       sql

Signature      accepts-n-syntax &key database => nil-t-required

Arguments      database A database object (default *default-data-base*).
Values

nil-t-required  t, nil or :required.

Description

The function accepts-n-syntax returns nil-t-required to indicate the behavior of the N syntax for string literals in the database specified by database. (The N syntax prefixes a string literal by the character N.)

nil-t-required can be one of:

nil  the database will give an error.
T  the database accepts it but does not require it.
:required  the database requires it (at least in some cases).

Currently, Microsoft SQL Server (which can be used via ODBC) is the only supported database that requires the N syntax for non-ASCII strings. SQLite and Microsoft Access (via ODBC) give errors. The other supported databases accept the syntax but do not need it.

If you use the “Symbolic SQL syntax” on page 358, then you can use the string pseudo-operator, which is described in “SQL string literals” on page 366 to obtain the correct syntax.

See also

“SQL string literals” on page 366
“Using non-ASCII strings on Microsoft SQL Server” on page 391
string-needs-n-prefix
string-prefix-with-n-if-needed

add-sql-stream

Function

Summary

Adds a stream to the broadcast list for SQL commands or results traffic.

Package

sql
This chapter applies to the Enterprise Edition only

Signature

```
add-sql-stream stream &key type database => added-stream
```

Arguments

- `stream`: A stream, or `t`.
- `type`: A keyword.
- `database`: A database.

Values

- `added-stream`: The argument `stream`.

Description

The function `add-sql-stream` adds the stream `stream` to the list of streams which receive SQL commands traffic or results traffic.

To add `*standard-output*` to the list, pass `stream t`.

The argument `type` is one of `:commands`, `:results` or `:both`, and determines whether a stream for commands traffic, results traffic, or both is added.

The argument `type` has a default value of `:commands`. The `database` is the value of `*default-database*` by default.

See also

- `*default-database*`
- `delete-sql-stream`
- `list-sql-streams`
- `sql-recording-p`
- `sql-stream`
- `start-sql-recording`
- `stop-sql-recording`

**attribute-type**

Function

Summary

Returns the type of an attribute.

Package

`sql`

Signature

```
attribute-type attribute table &key database owner => datatype
```
Arguments

- **table**: A table.
- **attribute**: An attribute from `table`.
- **database**: A database.
- **owner**: `nil`, `:all` or a string.

Values

- **datatype**: A keyword or list denoting a vendor-specific type.

Description

The function `attribute-type` returns the type of the attribute specified by `attribute` in the table given by `table`. The database, in which `table` is found, has a default value of `*default-database*`.

- If `owner` is `nil`, only user-owned attributes are considered. This is the default.
- If `owner` is `:all`, all attributes are considered.
- If `owner` is a string, this denotes a username and only attributes owned by `owner` are considered.

`datatype` demotes a vendor-specific type. Examples in a MS Access database are `:integer`, `:longchar` and `:datetime`. When `datatype` is a list, the second element is the length of the type, for example `(:varchar 255)`.

Example

To print the type of every attribute in the database, do

```lisp
(loop for tab in
  (sql:list-tables)
  do
    (loop for att in
      (sql:list-attributes tab)
      do
        (format t "-T*Table ~S Attribute ~S Type ~S" 
          tab att
          (sql:attribute-type att tab))))
```
This chapter applies to the Enterprise Edition only

See also
*default-database*
list-attribute-types
list-attributes

**cache-table-queries**

*Function*

**Summary**
Controls the caching of attribute type information.

**Package**
sql

**Signature**
cache-table-queries table &key database action

**Arguments**
table A string naming a table, :default or t.
database A database.
action t, nil or :flush.

**Description**
The function `cache-table-queries` provides per-table control on the caching in a particular database connection of attribute type information using during update operations.

If `table` is a string, it is the name of the table for which caching is to be altered. If `table` is t, then the `action` applies to all tables. If `table` is :default, then the default caching action is set for those tables which do not have an explicit setting.

`database` specifies the database connection, its default value is the value of *default-database*.

`action` specifies the caching action. The value t means cache the attribute type information. The value nil means do not cache the attribute type information. If `table` is :default, the setting applies to all tables which do not have an explicit setup.

The value :flush means remove any existing cache for `table` in `database`, but continue to cache.
**cache-table-queries** should be called with *action*: flush when the attribute specifications in *table* have changed.

See also

*cache-table-queries-default*
*default-database*

**cache-table-queries-default***

*Variable*

Package       sql
Initial value nil
Description   The variable *cache-table-queries-default* provides the default attribute type caching behavior. It allowed values are as described for the *action* argument of cache-table-queries.

See also  cache-table-queries

**commit***

*Function*

Summary       Commits changes made to a database.
Package       sql
Signature     commit &key database => nil
Arguments     database A database.
Values        nil
Description   The function commit commits changes made to the database specified by database, which is *default-database* by default.
This chapter applies to the Enterprise Edition only

Example

This example changes records in a database, and uses `commit` to make those changes permanent.

```lisp
(insert-records :into [emp]
    :attributes '(x y z)
    :values '(a b c))
(update-records [emp]
    :attributes [dept]
    :values 50
    :where [= [dept] 40])
(delete-records :from [emp]
    :where [> [salary] 300000])
(commit)
```

See also

- *default-database*
- rollback
- with-transaction

connect

Function

Summary

Opens a connection to a database.

Package

`sql`

Signature

```lisp
connect connection-spec &key if-exists database-type interface
    name encoding signal-rollback-errors default-table-type default-table-extra-options date-string-format sql-mode prefetch-rows-number prefetch-memory sqlite-keywords => database
```

Arguments

- `connection-spec` The connection specifications.
- `if-exists` A keyword.
- `database-type` A database type.
- `interface` A displayed CAPI element, or `nil`.
- `name` A Lisp object.
- `encoding` A keyword naming an encoding.
- `signal-rollback-errors`
nil, the keyword :default, or a function
designator.

default-table-type A string, the keyword
:support-transactions, or nil.

default-table-extra-options
A string or nil.

date-string-format A string, or the keyword :standard, or nil.

sql-mode A string or nil.

prefetch-rows-number
An integer or the keyword :default.

prefetch-memory An integer or the keyword :default.

sqlite-keywords A property list of keywords and values specific to SQLite.

Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>A database.</td>
</tr>
</tbody>
</table>

Description

The function connect opens a connection to a database of type database-type.

The allowed values for database-type are :odbc, :odbc-driver, :mysql, :postgresql, :oracle8 and :oracle, though not all of these are supported on some platforms. See “Supported databases” on page 333 for details of per-platform database support.

The default for database-type is the value of *default-database-type*.

connect sets the variable *default-database* to an instance of the database opened, and returns that instance.

If connection-spec is a list it is interpreted as a plist of keywords and values. Some of the keywords are database-type specific: see “Connecting to Oracle”, “Connecting to ODBC”, “Connecting to MySQL” or “Connecting to PostgreSQL” as appropriate.
This chapter applies to the Enterprise Edition only

General connection-spec keywords are:

:username
User name

:password
Password

:connection
A specification of the connection. In general, this is supposed to be sufficient information (other than the username and password) to open a connection. The precise meaning varies according to the database-type.

If connection-spec is a string, it is interpreted canonically as:

username/password@connection

where connection can be omitted along with the '@' in cases when there is a default connection, password can be omitted along with the preceding '/', and username can be omitted if there is a default user. For example, if you have an Oracle user matching the current Unix username and that does not need a password to connect, you can call

(connect "/")

Specific database-types may allow more elaborate syntax, but conforming to the pattern above. See the section “Initialization” on page 334 for details.

Additionally for database-types :odbc and :odbc-driver, if connection-spec does not include the '@' character then the string is interpreted in a special way, for backward compatibility with LispWorks 4.4 and earlier versions. See the section “Connecting to ODBC” on page 339 for details.

name can be passed to explicitly specify the name of the connection. If name is supplied then it is used as-is for the connection name. Therefore it can be found by another call to connect and calls to find-database. Connection names are compared with eql. If name is not supplied, then a unique database name is constructed from connection-spec and a counter.
If *name* is supplied then existing connections are found by comparing their name with *name* and then *if-exists* modifies the behavior of *connect* as follows:

- **:new**: Makes a new connection even if connections to the same database already exist.
- **:warn-new**: Makes a new connection but warns about existing connections.
- **:error**: Makes a new connection but signals an error for existing connections.
- **:warn-old**: Selects an existing connection if there is one (and warns), or makes a new connection.
- **:old**: Selects an existing connection if there is one, or makes a new one.

The default value of *if-exists* is the value of *connect-if-exists*.

*interface* is used if *connect* needs to display a dialog to ask the user for username and password. If *interface* is a CAPI element, this is used. If *interface* is any other value (the default value is *nil*), and *connect* is called in a process which is associated with a CAPI interface, then this CAPI interface is used. *interface* has been added because dialogs asking for passwords can fail otherwise. This depends on the driver that the datasource uses: the problem has only been observed using MS SQL on Microsoft Windows.

*encoding* specifies the encoding to use in the connection. The value should be a keyword naming an acceptable encoding, or *nil* (the default). The value *unicode* is accepted for all *database-type* specific:

- **:mysql**: If *encoding* is *nil* or *:default* then the encoding is chosen according to the default character set of the connection (if available)
and if that fails the encoding :utf-8 is used. The other recognized values of encoding are :unicode, :utf-8, :ascii, :latin-1, :euc and :sjis.

:unicode uses :utf-8 internally.

:postgresql

If encoding is nil or :default LispWorks does not set anything in the connection. If the connection character set is SQL_ASCII, LispWorks uses :latin-1 to convert to and from Lisp strings, otherwise it uses :utf-8.

If encoding is one of the keywords listed below, LispWorks uses it as the external format for converting to and from Lisp strings, and LispWorks also sets the connection character set to the corresponding string:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>character-set</th>
<th>alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>:utf-8</td>
<td>UTF-8</td>
<td></td>
</tr>
<tr>
<td>:latin-1</td>
<td>SQL_ASCII</td>
<td></td>
</tr>
<tr>
<td>:ascii</td>
<td>SQL_ASCII</td>
<td></td>
</tr>
<tr>
<td>:gbk</td>
<td>GBK</td>
<td></td>
</tr>
<tr>
<td>:euc-jp</td>
<td>EUCJP</td>
<td>:euc</td>
</tr>
<tr>
<td>:sjis</td>
<td>SJIS</td>
<td>:shift-jis</td>
</tr>
</tbody>
</table>

An alias maps to the corresponding keyword.

In addition, encoding can be a string or a cons of a keyword and a string. If it is a string LispWorks uses :utf-8 as the external format, and sets the connection character set to the string. If it is a cons, the keyword (the car) is used as the external format, and the string (cdr) is used to set the character set.

See "character set support" in the PostgresQL manual for known character sets.

:oracle

The only recognized values of encoding are nil and :unicode.
The SQL Package

This chapter applies to the Enterprise Edition only

:oracle8

encoding is ignored.

:odbc or :odbc-driver

The valid values of encoding are :unicode or nil. When encoding is nil it uses the default multibyte encoding.

:sqlite

If encoding is :default, :unicode or :utf-8 then UTF-8 is used (by calling the C function sqlite3_open_v2). If encoding is :utf-16 or :utf-16-native, then UTF-16 in the native byte order is used (by calling the C function sqlite3_open16). It is not obvious in what circumstances UTF-16 is better and it is made available only because the underlying library supports it.

signal-rollback-errors controls what happens when an attempted rollback causes an error, for databases that do not support rollback properly (for example MySQL with the default settings). For database-types other than :mysql signal-rollback-errors is ignored and such an error is always signaled. For database-type :mysql signal-rollback-errors is interpreted as follows:

nil Ignore the error.

:default If default-table-type is :support-transac-
tions, "innodb" or "bdb", then rollback errors are signaled. Otherwise rollback errors are not signaled.

Function designator

The function signal-rollback-errors should take two arguments: the database object and a string (for an error message). The function is called when a rollback signaled an error.

The default value of signal-rollback-errors is :default.
default-table-type specifies the default value of the :type argument to create-table. See create-table for details. The default value of default-table-type is nil.

default-table-extra-options specifies the default value of the :extra-options argument to create-table. See create-table for details. The default value of default-table-extra-options is nil.

date-string-format specifies which format to use to represent dates. If the value is a string, it should be appropriate for the database-type. The value :standard means that the standard SQL date format is used. If the value is nil (the default), then the date format is not changed. Currently only database-type :oracle uses the value of date-string-format, and in this case it must be a valid date format string for Oracle.

sql-mode specifies the mode of the SQL connection for database-type :mysql. By default (that is, when sql-mode is not supplied) connect sets the mode of the connection to ANSI, by executing this statement:

   "set sql_mode='ansi'"

sql-mode can be supplied as nil, in which case no statement is executed. Otherwise it should be a string which is a valid setting for sql_mode, and then connect executes the statement:

   set sql_mode='sql-mode'

When database-type is not :mysql, sql-mode is ignored.

prefetch-rows-number and prefetch-memory are used when database-type is :oracle, and specify the amount of data to prefetch when performing queries. prefetch-rows-number is the number of rows to prefetch, with default value 100. prefetch-memory is the maximum number of bytes to prefetch, with default value #x100000. prefetch-rows-number and prefetch-memory can both also have the value :default, which allows the database to choose the amount to prefetch.
sqlite-keywords is used only when connecting to SQLite (database-type is :sqlite) and is ignored otherwise. See “SQLite connection keywords” on page 344 for more details.

**Notes**

All the Common SQL functions that accept the keyword argument :database use **find-database** to find the database if the given value is not a database. Therefore these functions can now find only databases that that were opened with an explicit name:

```lisp
(connect ... :name name ...) 
```

**Compatibility notes**

LispWorks 4.4 (and previous versions) use **connection-spec** passed to **connect** as the database name. **connect** checks whether a connection with this name already exists (according to the value of **if-exists**). **find-database** can be used to find a database using this name.

LispWorks 5.0 (and later versions) does not use **connection-spec** as the name. Instead, by default it generates a name from the **connection-spec**. The name is intended to be unique (by including a counter). Thus normally **connect** will not find an existing connection even if it is called again with identical value of **connection-spec**.

**Example**

The following example connects LispWorks to the **info** database.

```lisp
(connect "info")
```

The next example connects to the ODBC database **personnel** using the username "admin" and the password "secret".

```lisp
(connect "personnel/admin/secret" :database-type :odbc)
```

The next example opens a connection to MySQL which treats quotes as in ANSI but does not set other ANSI features:

```lisp
(sql:connect "me/mypassword/mydb"
 :sql-mode "ANSI_QUOTES")
```
This chapter applies to the Enterprise Edition only

See also
- *default-database*
- *default-database-type*
- connected-databases
- *connect-if-exists*
- database-name
- disconnect
- find-database
- reconnect
- status

*connect-if-exists*  

**Variable**

**Summary** The default value for the *if-exists* keyword of the *connect* function.

**Package** sql

**Initial value** :error

**Description** The variable *connect-if-exists* is the default value for the *if-exists* keyword of the *connect* function. It can take the following values:

- :new
  - Instructs *connect* to make a new connection even if connections to the same database already exist.

- :warn-new
  - Instructs *connect* to make a new connection but warn about existing connections.

- :error
  - Instructs *connect* to make a new connection but signal an error for existing connections.

- :warn-old
  - Instructs *connect* to select an old connection if one exists (and warns) or make a new one.

- :old
  - Instructs *connect* to select an old connection if one exists or make a new one.
See also  connect

**connected-databases**  
*Function*

**Summary** Returns a list of connected databases.

**Package** sql

**Signature** connected-databases => database-list

**Arguments** None.

**Values** database-list A list of connected databases.

**Description** The function **connected-databases** returns a list of the databases LispWorks is connected to.

See also  connect  
disconnect  
status  
find-database  
database-name

**create-index**  
*Function*

**Summary** Creates an index for a table.

**Package** sql

**Signature** create-index name &key on unique attributes database =>

**Arguments** name The name of the index.  
on The name of a table.  
unique A boolean.
This chapter applies to the Enterprise Edition only

\[ attributes \] A list of attributes.
\[ database \] A database.

**Values** None.

**Description** The function `create-index` creates an index called `name` on the table specified by `on`. The attributes of the table to index are given by `attributes`. Setting `unique` to `t` includes `UNIQUE` in the SQL index command, specifying that the columns indexed must contain unique values.

The default value of `unique` is `nil`. The default value of `database` is `*default-database*`.

**Example**

```lisp
(create-index [manager]
  :on [emp] :unique t :attributes '([ename] [sal]))
```

**See also**

`*default-database*`

`drop-index`

`create-table`

---

**create-table**

**Function**

**Summary** Creates a table.

**Package** `sql`

**Signature**

```
create-table name description &key database type extra-options
```

**Arguments**

- `name` The name of the table.
- `description` The table properties.
- `database` A database.
- `type` A string or the keyword `:support-transactions`, or `nil`.
- `extra-options` A string or `nil`.
The function `create-table` creates a table called `name` and defines its columns and other properties with `description`. The argument `description` is a list containing lists of attribute-name and type information pairs.

The default value of `database` is `*default-database*`.

`type` and `extra-options` are treated in a `database-type` specific way. Currently only `database-type :mysql` uses these options, as follows.

If `type` is not supplied, it defaults to the value (if any) of `default-table-type` that was supplied to `connect`. If `extra-options` is not supplied, it defaults to the value (if any) of `default-table-extra-options` that was supplied to `connect`.

`type`, if non-nil, is used as argument to `TYPE` in the SQL statement:

```
create table MyTable (column-specs) TYPE = type
```

except that if `type` is `:support-transactions` then `create-table` will attempt to make tables that support transactions, by using the type `innodb`.

`extra-options` (if non-nil) is appended in the end of this SQL statement.

When `database-type` is not `:mysql`, `type` and `extra-options` are ignored.

**Example**

The following code:

```
(create-table [manager]
  '(([[id] (char 10) not-null)
    ([salary] (number 8 2))))
```

is equivalent to the following SQL:

```
CREATE TABLE MANAGER
  (ID CHAR(10) NOT NULL, SALARY NUMBER(8,2))
```
This chapter applies to the Enterprise Edition only

See also

connect
*default-database*
drop-table

create-view

Function

Summary Creates a view using a specified query.

Package sql

Signature create-view name &key as column-list with-check-option
database =>

Arguments

name The view to be created.
as A SQL query statement.
column-list A list.
with-check-option A boolean.
database A database.

Values None.

Description The function create-view creates a view called name using
the as query and the optional column-list and with-check-option. The column-list argument is a list of columns to add to
the view. The with-check-option adds WITH CHECK OPTION to
the resulting SQL.

The default value of with-check-option is nil. The default
value of database is *default-database*.

Example This example creates the view manager with the records in
the employee table whose department is 50.
The function `create-view-from-class` creates a view in a database based on a class that defines the view. The argument `database` has a default value of `*default-database*`.

Parameters
- `class`: A class.
- `database`: A database.

Description

```
(create-view [manager] :as [select [*]
    :from [emp]
    :where [= [dept] 50]])
```

See also
- `create-index`
- `create-table`
- `*default-database*`
- `drop-view`

**database-name**

The function `database-name` returns the name of a database.

Summary

```
Returns the name of a database.
```
This chapter applies to the Enterprise Edition only

Package sql

Signature database-name database => connection

Arguments database A database.

Values connection A string.

Description The function database-name returns the name of the database specified by database.

See also connect disconnect connected-databases find-database status

decode-to-db-standard-date
decode-to-db-standard-timestamp

Functions

Summary Convert Lisp universal time to standard SQL DATE and TIMESTAMP.

Package sql

Signature decode-to-db-standard-date universal-time &key stream quoted => date
decode-to-db-standard-timestamp universal-time &key stream quoted => timestamp

Arguments universal-time A universal time.
stream nil, t, or an output stream.
quoted A boolean.

Values date A string or nil.
timestamp A string or nil.

Description The functions `decode-db-standard-date` and `decode-db-standard-timestamp` take a Lisp universal time and convert it to a SQL DATE or TIMESTAMP respectively.

The format of the date is YYYY-MM-DD.

The format of the timestamp is YYYY-MM-DD HH:MM:SS.

`stream` is interpreted as in `cl:format`. If `stream` is `nil` then the string representing the DATE or TIMESTAMP is returned, otherwise the string is written to the stream and `nil` is returned. The default value of `stream` is `nil`.

When `quoted` is true, the date or timestamp is quoted (by single quote). This is useful when these functions are used while building a SQL command string, and the result should be interpreted as a string. The default value of `quoted` is `nil`.

See also `encode-db-standard-date`  
`encode-db-standard-timestamp`  
`connect`  
“Working with date fields” on page 370

**encode-db-standard-date**  
**encode-db-standard-timestamp**  

Functions

| Summary | Convert standard SQL DATE and TIMESTAMP to Lisp universal time. |
| Package | sql |
| Signature | `encode-db-standard-date date-string => result`  
`encode-db-standard-timestamp timestamp-string => result` |
| Arguments | `date-string` A string. |
timestamp-string  A string.

Values  

*result*  

A Lisp universal time or *nil*.

Description  

The functions `encode-db-standard-date` and `encode-db-standard-timestamp` interpret their argument as a DATE or TIMESTAMP and return the corresponding universal time.

date-string  must be a string of length at least 10, where the first 10 characters specify a DATE, that is have the format YYYY-MM-DD.

timestamp-string  must be a string of length at least 19, where the first 19 characters specify a TIMESTAMP, that is have the format YYYY-MM-DD HH:MM:SS.

`encode-db-standard-date` and `encode-db-standard-timestamp` do not actually check the separators between the numeric values, so the hyphens, space and colons can each be replaced by any character. Both functions return *nil* if the argument is not correct.

See also  

`decode-to-db-standard-date`  
`decode-to-db-standard-timestamp`  
`connect`  
“Working with date fields” on page 370

---

*default-database*  

**Variable**

Summary  

The default database in database operations.

Package  

sql

Initial value  

*nil*

Description  

The variable *default-database* is set by `connect` and specifies the default database to be used for database operations.
See also connect

**default-database-type**

*Variable*

Summary Specifies the default type of database.

Package sql

Initial value nil

Description The variable *default-database-type* specifies the default type of database. You can set this or it is initialized by the initialize-database-type function.

LispWorks supports the values shown in “Supported databases” on page 333.

See also initialize-database-type

**default-update-objects-max-len**

*Variable*

Summary The default maximum number of objects supplying data for a query when updating remote joins.

Package sql

Initial value nil

Description The variable *default-update-objects-max-len* provides the default value of the max-len argument in the function update-objects-joins.

See also update-objects-joins
This chapter applies to the Enterprise Edition only

**def-view-class**  
*Macro*

**Summary** Extends the syntax of *defclass* to allow specified slots to be mapped onto the attributes of database views.

**Package** sql

**Signature**  
def-view-class name superclasses slots &rest class-options => class

**Arguments**  
*name*  
A class name.

*superclasses*  
The superclasses of the class to be created.

*slots*  
The slot definitions of the new class.

*class-options*  
The class options of the new class.

**Values**  
class  
The defined class.

**Slot Options**  
The slot options for *def-view-class* are :db-kind and :db-info. In addition the slot option :type is treated specially for View Classes.

 :db-kind may be one of :base, :key, :join, or :virtual. The default is :base. Each value is described below:

 :base  
This indicates that this slot corresponds to an ordinary attribute of the database view. You can name the database attribute by using the keyword :column. By default, the database attribute is named by the slot.

 :key  
This indicates that this slot corresponds to part of the unique key for this view. A :key slot is also a :base slot. All View Classes must have :key fields that uniquely distinguish the instances, to maintain object identity.
To specify a key which spans multiple slots, each of the slots should have `:db-kind :key`. The underlying requirement is that tuples of the form `(key1 ... keyN)` are unique. The `:db-kind :key` slots do not need to be keys in the table.

`:join` This indicates that this slot corresponds to a join. A slot of this type will contain View Class objects.

`:virtual` This indicates that this slot is an ordinary CLOS slot not associated with a database column.

A join is defined by the slot option `:db-info`, which takes a list. Items in the list may be:

`:join-class class-name`

This is the class to join on.

`:home-key slot-name`

This is the slot of the defining class to be a subject for the join. The argument `slot-name` may be an element or a list of elements, where elements can be symbols, `nil`, strings, integers or floats.

`:foreign-key slot-name`

This is the name of the slot of the `:join-class` to be a subject for the join. The `slot-name` may be an element or a list of elements, where elements can be symbols, `nil`, strings, integers or floats.
This chapter applies to the Enterprise Edition only

[target-slot target-slot]

This is the name of a :join slot in :join-class. This is optional and is only specified if you want the defining slot to contain instances of this target slot as opposed to those of :join-class. The actual behavior depends on the value of set. An example of its usage is when the :join-class is an intermediate class and you are really only interested in it as a route to the :target-slot.

[retrieval retrieval-time]

retrieval-time can be :deferred, which defers filling this slot from the database until the slot itself is accessed. This is the default value.

retrieval-time can alternatively be :immediate, which generates the join SQL for this slot whenever a query is generated on the class. In other words, this is an intermediate class only, which is present for the purpose of joining two entities of other classes together. When retrieval-time is :immediate, then set is nil.

[set set]

When set is t and target-slot is defined, the slot will contain a list of pairs (target-value join-instance) where target-value is the value of the target slot and join-instance is the corresponding instance of the join class.

When set is t and target-slot is undefined, the slot will contain a list of instances of the join class.
When \( \text{set} \) is \text{nil} the slot will contain a single instance.

The default value of \( \text{set} \) is \text{t}.

The syntax for \text{:home-key} and \text{:foreign-key} means that an object from a join class will only be included in the join slot if the values from \text{home-key} are \text{equal} to the values in \text{foreign-key}, in order. These values are calculated as follows: if the element in the list is a symbol it is taken to be a slot name and the value of the slot is used, otherwise the element is taken to be the value. See the second example below.

The \text{:type} slot option is treated specially for View Classes. There is a need for stringent type-checking in View Classes because of the translation into database data, and therefore \text{:type} is mandatory for slots with \text{:db-kind :base} or \text{:key}.

Some methods are provided for type checking and type conversion. For example, a \text{:type} specifier of (\text{string 10}) in SQL terms means allow a character type value with length of less than or equal to 10. The following Lisp types are accepted for \text{type}, and correspond to the SQL type shown:

\[
\begin{align*}
\text{(string n)} & \quad \text{CHAR(n)} \\
\text{integer} & \quad \text{INTEGER} \\
\text{(integer n)} & \quad \text{INTEGER(n)} \\
\text{float} & \quad \text{FLOAT} \\
\text{(float n)} & \quad \text{FLOAT(n)} \\
\text{sql:universal-time} & \quad \text{TIMESTAMP}
\end{align*}
\]

\text{Class Options} \text{def-view-class} recognizes the following class options in addition to the standard class options defined for \text{defclass}:

\[
\begin{align*}
\text{(:base-table table-name)}
\end{align*}
\]
This chapter applies to the Enterprise Edition only

The slots of the class name will be read from the table table-name. If you do not specify the :base-table option, then table-name defaults to the name of the class.

Description

The macro def-view-class creates a class called name which maps onto a database view. Such a class is called a View Class.

The macro def-view-class extends the syntax of defclass to allow special base slots to be mapped onto the attributes of database views (presently single tables). When a select query that names a View Class is submitted, then the corresponding database view is queried, and the slots in the resulting View Class instances are filled with attribute values from the database.

If superclasses is nil then standard-db-object automatically becomes the superclass of the newly-defined View Class. If superclasses is nil, it must include standard-db-object.

Examples

The following example shows a class corresponding to the traditional employees table, with the employee’s department given by a join with the departments table.
(def-view-class employee (standard-db-object)
  ((employee-number :db-kind :key
    :column empno
    :type integer)
   (employee-name :db-kind :base
    :column ename
    :type (string 20)
    :accessor employee-name)
   (employee-department :db-kind :base
    :column deptno
    :type integer
    :accessor employee-department)
   (employee-job :db-kind :base
    :column job
    :type (string 9))
   (employee-manager :db-kind :base
    :column mgr
    :type integer)
   (employee-location :db-kind :join
     :db-info (:join-class department
               :retrieval :deferred
               :set nil
               :home-key employee-department
               :foreign-key department-number
               :target-slot department-loc
               :accessor employee-location))
  (:base-table emp))

The following example illustrates how elements or lists of
elements can follow :home-key and :foreign-key in the :db-info slot option.

(def-view-class flex-schema ()
  ((name :type (string 8) :db-kind :key)
   (description :type (string 256))
   (classes :db-kind :join
     :db-info (:home-key name
               :foreign-key schema-name
               :join-class flex-class
               :retrieval :deferred))
  (:base-table flex_schema))
(def-view-class flex-class ()
  ((schema-name :type (string 8) :db-kind :key
    :column schema_name)
   (name                :type (string 32) :db-kind :key)
   (base-name           :type (string 64) :column base_name)
   (super-classes       :db-kind :join
    :db-info (:home-key
      (schema-name name)
      :foreign-key (schema-name class-name)
      :join-class flex-superclass
      :retrieval :deferred))
   (schema            :db-kind :join
    :db-info (:home-key schema-name
      :foreign-key name
      :join-class flex-schema
      :set nil))
   (properties       :db-kind :join
    :db-info (:home-key (schema-name name"
      :foreign-key (schema-name class-name slot-name)
      :join-class flex-property
      :retrieval :deferred))
   (:base-table flex_class))

(def-view-class flex-slot ()
  ((schema-name :type (string 8) :db-kind :key
    :column schema_name)
   (class-name :type (string 32) :db-kind :key
    :column class_name)
   (name        :type (string 32) :db-kind :key)
   (class       :db-kind :join
    :db-info (:home-key (schema-name class-name)
      :foreign-key (schema-name name)
      :join-class flex-class
      :set nil))
   (properties       :db-kind :join
    :db-info (:home-key
      (schema-name class-name slot-name)
      :foreign-key (schema-name class-name slot-name)
      :join-class flex-property
      :retrieval :deferred))
   (:base-table flex_slot))

This chapter applies to the Enterprise Edition only
(def-view-class flex-property ()
  (((schema-name :type (string 8) :db-kind :key
                  :column schema_name)
    (class-name :type (string 32) :db-kind :key
                :column class_name)
    (slot-name :type (string 32) :db-kind :key
               :column slot_name)
    (property :type (string 32) :db-kind :key))
  (values :db-kind :join
          :db-info (:home-key
                    (schema-name class-name
                    slot-name property)
                   :foreign-key
                    (schema-name class-name
                    slot-name property)
                   :join-class flex-property-value
                   :retrieval :deferred)))
 (:base-table flex_property))

(def-view-class flex-property-value ()
  (((schema-name :type (string 8) :db-kind :key
                  :column schema_name)
    (class-name :type (string 32) :db-kind :key
                :column class_name)
    (slot-name :type (string 32) :column slot_name)
    (property :type (string 32) :db-kind :key)
    (order :type integer)
    (value :type (string 128)))
 (:base-table flex_property_value))

See also
create-view-from-class
delete-instance-records
drop-view-from-class
standard-db-object
update-record-from-slot
update-records-from-instance

delete-instance-records

Generic Function

Summary
Deletes records corresponding to View Class instances.

Package sql
This chapter applies to the Enterprise Edition only

**delete-instance-records**

**Signature**

`delete-instance-records instance =>`

**Arguments**

`instance` An instance of a View Class.

**Values**

None.

**Description**

The function `delete-instance-records` deletes the records represented by `instance` from the database associated with it. If `instance` has no associated database, `delete-instance-records` signals an error.

**See also**

update-records
update-records-from-instance

---

**delete-records**

**Function**

**Summary**

Deletes rows from a database table.

**Package**

`sql`

**Signature**

`delete-records &key from where database =>`

**Arguments**

`from` A database table.

`where` A SQL conditional statement.

`database` A database.

**Values**

None.

**Description**

The function `delete-records` deletes rows from a table specified by `from` in which the `where` condition is true. The argument `database` specifies a database from which the records are to be removed, and defaults to `*default-database*`. 
See also *default-database*
insert-records
update-records

default-database

**delete-sql-stream**

*default-database*

## Function

**Summary**
Deletes a stream from the broadcast list for SQL commands or results traffic.

**Package**
sql

**Signature**
delete-sql-stream stream &key type database => deleted-stream

**Arguments**
- stream A stream or t.
- type A keyword.
- database A database.

**Values**
- deleted-stream The argument stream.

**Description**
The function `delete-sql-stream` deletes the stream `stream` from the list of streams which receive SQL commands or results traffic.

To remove *standard-output* from the list, pass `stream t`.

The keyword type is :commands, :results or :both. It determines whether a stream for SQL commands traffic, results traffic, or both is deleted.

The default value of type is :commands. The default value for database is the value of *default-database*.

See also
- add-sql-stream
- list-sql-streams
- sql-recording-p
- sql-stream
This chapter applies to the Enterprise Edition only

```
start-sql-recording
stop-sql-recording
```

**destroy-prepared-statement**  
*Function*

**Summary**  
Destroys a prepared-statement and frees its resources.

**Package**  
sql

**Signature**  
`destroy-prepared-statement prepared-statement => nil`

**Arguments**  
`prepared-statement`

A prepared-statement.

**Description**  
The function `destroy-prepared-statement` destroys the prepared-statement prepared-statement and frees its resources. It should be called before closing the database associated with prepared-statement. A destroyed prepared-statement can be reused by calling `set-prepared-statement-variables` with a new database.

`destroy-prepared-statement` always returns nil.

**See also**  
prepare-statement
set-prepared-statement-variables

**disable-sql-reader-syntax**  
*Function*

**Summary**  
Turns off square bracket syntax.

**Package**  
sql

**Signature**  
`disable-sql-reader-syntax =>`

**Arguments**  
None.
The SQL Package

This chapter applies to the Enterprise Edition only

<table>
<thead>
<tr>
<th>Values</th>
<th>None.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The function <code>disable-sql-reader-syntax</code> turns off square bracket syntax and sets state so that <code>restore-sql-reader-syntax-state</code> will make the syntax disabled if it is consequently enabled.</td>
</tr>
</tbody>
</table>
| See also | `enable-sql-reader-syntax`  
`locally-disable-sql-reader-syntax`  
`locally-enable-sql-reader-syntax`  
`restore-sql-reader-syntax-state` |

### `disconnect`  

**Function**

**Summary**
Closes a connection to a database.

**Package**
`sql`

**Signature**
`disconnect &key database error => success`

**Arguments**
- `database` A database.
- `error` A boolean.

**Values**
- `success` A boolean.

**Description**
The function `disconnect` closes a connection to a database specified by `database`. If successful, `success` is `t` and if only one other connection exists, `*default-database*` is reset.

The default value for `database` is `*default-database*`. If `database` is a database object, then it is used directly.

Otherwise, the list of connected databases is searched to find one with database as its connection specifications (see `connect`). If no such database is found, then if `error` and `database` are both non-nil an error is signaled, otherwise `disconnect` returns `nil`. 
This chapter applies to the Enterprise Edition only

Example

(disconnect :database "test")

See also

connect
connected-databases
database-name
*default-database*
find-database
reconnect
status

**do-query**

Macro

Summary

Repeatedly binds a set of variables to the results of a query, and executes a body of code using the bound variables.

Package

sql

Signature

**do-query** (&rest **args** **query** \&key **database** **not-inside-transaction** **get-all**) **&body** **body** =>

Arguments

**args**

A set of variables.

**query**

A database query or a **prepared-statement** containing a query.

**database**

A database.

**not-inside-transaction**

A generalized boolean.

**get-all**

A generalized boolean.

**body**

A Lisp code body.

Values

None.

Description

The macro **do-query** repeatedly executes **body** within a binding of **args** on the attributes of each record resulting from **query**. **do-query** returns no values.
The default value of database is *default-database*.

not-inside-transaction and get-all may be useful when fetching many records through a connection with database-type :mysql. Both of these arguments have default value nil. See the section “Special considerations for iteration functions and macros” on page 375 for details.

Example

The following code repeatedly binds the result of selecting an entry in ename from the table emp to the variable name, and then prints name using the Lisp function print.

```
(do-query ((name) [select [ename] :from [emp]])
  (print name))
```

See also

loop
map-query
prepare-statement
query
select
simple-do-query

drop-index

Function

Summary

Deletes an index from a database.

Package

sql

Signature

drop-index index &key database =>

Arguments

index The name of an index.
database A database.

Values

None.

Description

The function drop-index deletes index from database.

The default value of database is *default-database*.
drop-table  

Function  

Summary  

Deletes a table from a database.

Package  

sql

Signature  

drop-table table &key database =>

Arguments  

table The name of a table.  
database A database.

Values  

None.

Description  

The function drop-table deletes table from a database.

The default value of database is *default-database*.

See also  

create-table  

*default-database*

drop-view  

Function  

Summary  

Deletes a view from a database.

Package  

sql

Signature  

drop-view view &key database =>

Arguments  

view A view.  
database A database.

Values  

None.
The SQL Package

This chapter applies to the Enterprise Edition only.

Description
The function `DROP VIEW` deletes `view` from `database`.
The default value of `database` is `*default-database*`.

Notes
`DROP VIEW` is not implemented in MS Access SQL, so `drop-view` does not work with that database. Use `drop-table` instead.

See also
`create-view`  
`*default-database*`  
`drop-index`  
`drop-table`

**drop-view-from-class**

Function

Summary
Deletes a view from a database based on a class defining the view.

Package
`sql`

Signature
`drop-view-from-class class &key database =>`

Arguments
`class` A class.  
`database` A database.

Values
None.

Description
The function `drop-view-from-class` deletes a view or base table from `database` based on `class` which defines that view. The argument `database` has a default value of `*default-database*`.

See also
`create-view-from-class`  
`*default-database*`  
`drop-view`
This chapter applies to the Enterprise Edition only

**enable-sql-reader-syntax**

**Function**

**Summary**

Turns on square bracket SQL syntax.

**Package**

sql

**Signature**

`enable-sql-reader-syntax =>`

**Arguments**

None.

**Values**

None.

**Description**

The function `enable-sql-reader-syntax` turns on square bracket syntax and sets the state so that `restore-sql-reader-syntax-state` will make the syntax enabled if it is subsequently disabled.

**See also**

`disable-sql-reader-syntax`

`locally-disable-sql-reader-syntax`

`locally-enable-sql-reader-syntax`

`restore-sql-reader-syntax-state`

**execute-command**

**Function**

**Summary**

Executes a SQL expression.

**Package**

sql

**Signature**

`execute-command sql-exp &key database =>`

**Arguments**

`sql-exp` Any SQL statement other than a query.

`database` A database.

**Values**

None.
Description

The function `execute-command` executes the SQL command specified by `sql-exp` for the database specified by `database`, which has a default value of `*default-database*`. The argument `sql-exp` may be any SQL statement other than a query.

To run a stored procedure, pass an appropriate string. The call to the procedure needs to be wrapped in a PL/SQL `BEGIN END` pair, for example:

```
(sql:execute-command
 *BEGIN my_procedure(1, 'foo'); END;*)
```

See also

*default-database*

query

---

**find-database**

Function

Summary

Returns a database, given a database or database name.

Package

sql

Signature

`find-database database &optional errorp => database, count`

Arguments

- `database`: A string or a database.

Values

- `database`: A database.
- `count`: An integer.

Description

The function `find-database`, given a string `database`, searches amongst the connected databases for one matching the name `database`.

If there is exactly one such database, it is returned and the second return value `count` is 1. If more than one databases match and `errorp` is `nil`, then the most recently connected of
This chapter applies to the Enterprise Edition only

the matching databases is returned and count is the number
of matches. If no matching database is found and errorp is
nil, then nil is returned. If none, or more than one, match-
ing databases are found and errorp is true, then an error is sig-
naled.

If the argument database is a database, it is simply returned.

See also

connect
connected-databases
database-name
disconnect
status

initialize-database-type

Function

Summary

Initializes a database type.

Package

sql

Signature

initialize-database-type &key database-type => type

Arguments

database-type A database type.

Values

type A database type.

Description

The function initialize-database-type initializes a data-
base type by loading code and appropriate database libraries
according to the value of database-type. If *default-data-
base-type* is not initialized, this function initializes it. It
adds database-type to the list of initialized types. The initial-
ized database type is returned.

Example

The following example shows how to use initialize-
database-type to initialize the :odbc database type.
(require "odbc")
(in-package sql)
(setf *default-database-type* :odbc)
(initialize-database-type)
(print *initialized-database-types*)

The ODBC database type is now initialized, and connections can be made to ODBC databases.

See also

database-name

*initialized-database-types*

*default-database-type*

*initialized-database-types*

**Variable**

**Summary**
A list of initialized database types.

**Package**
sql

**Initial value**
nil

**Description**
The variable *initialized-database-types* contains a list of database types that have been initialized by calls to initialize-database-type.

See also
initialize-database-type

**insert-records**

**Function**

**Summary**
Inserts a set of values into a table.

**Package**
sql

**Signature**
insert-records &key into attributes values av-pairs query database

**Arguments**
into A database table.
This chapter applies to the Enterprise Edition only

values A list of values, or nil
attributes A list of attributes, or nil
av-pairs A list of two-element lists, or nil.
query A query expression, or nil.
database A database.

Values None.

Description The function insert-records inserts records into the table into.

The records created contain values for attributes (or av-pairs).
The argument values is a list of values. If attributes is supplied then values must be a corresponding list of values for each of the listed attribute names.

If av-pairs is non-nil, then both attributes and values must be nil.

If query is non-nil, then neither values nor av-pairs should be.
query should be a query expression, and the attribute names in it must also exist in the table into.

The default value of database is *default-database*.

Example In the first example, the Lisp expression

(insert-records :into [person]
  :values '("abc" "Joe" "Bloggs" 10000 3000 nil "plumber"))

is equivalent to the following SQL:

\[
\text{INSERT INTO PERSON}
\begin{align*}
\text{VALUES (}} & \text{'abc', 'Joe', } \\
& \text{ 'Bloggs', 10000, 3000, NULL, 'plumber')} \\
\end{align*}
\]

In the second example, the LispWorks expression

(insert-records :into [person]
  :attributes '(person_id income surname occupation)
  :values '(['aaa' 10 'jim' 'plumb'])

1571
is equivalent to the following SQL:

```
INSERT INTO PERSON
    (PERSON_ID, INCOME, SURNAME, OCCUPATION)
VALUES ('aaa', 10, 'jim', 'plumb')
```

The following example demonstrates how to use `:av-pairs`.

```
(insert-records :into [person] :av-pairs
    '((person_id "bbb") (surname "Jones")))
```

See also
- `*default-database*`
- `delete-records`
- `update-records`

---

**instance-refreshed**

*Generic Function*

**Summary**

Provides a hook for user code on View Class instance updates.

**Package**

`sql`

**Signature**

`instance-refreshed instance`

**Arguments**

`instance` An instance of a View Class.

**Values**

None.

**Description**

The function `instance-refreshed` is called inside `select` when its `refresh` argument is true and the instance `instance` has just been updated.

The supplied method on `standard-db-object` does nothing. If your application needs to take action when a View Class instance has been updated by

```
(select ... :refresh t)
```

then add an `instance-refresh` method specializing on your subclass of `standard-db-object`. 
This chapter applies to the Enterprise Edition only

See also  def-view-class  
          select

list-attribute-types  

Function

Summary  
Returns type information for a table’s attributes.

Package  sql

Signature  
list-attribute-types table &key database owner => result

Arguments  
table  A table.

database  A database.

owner  nil, :all or a string.

Values  
result  A list.

Description  
The function list-attribute-types returns type information for the attributes in the table given by table.

database has a default value of *default-database*.

If owner is nil, only user-owned attributes are considered. This is the default.

If owner is :all, all attributes are considered.

If owner is a string, this denotes a username and only attributes owned by owner are considered.

result is a list in which each element is a list (attribute datatype precision scale nullable). attribute is a string denoting the attribute name. datatype is the vendor-specific type as described in attribute-type. nullable is 1 if the attribute accepts the value NULL, and 0 otherwise.

Notes  
When using ODBC to connect to Access database, the nullable value is not reliable, at least on version 7.1. There seems to be
a bug in the driver. Using ODBC with other DBMS works as documented.

**Example**

To print the type of every attribute in the database, do

```
(loop for tab in
    (sql:list-tables)
        do
            (loop for type-info in
                (sql:list-attribute-types tab)
                    do
                        (format t "~&Table ~S Attribute ~S Type ~S"
                            tab
                            (first type-info)
                            (second type-info))))
```

**See also**

- attribute-type
- list-attributes

### list-attributes

**Function**

**Summary**

Returns a list of attributes from a table in a database.

**Package**

`sql`

**Signature**

`list-attributes table &key database owner => result`

**Arguments**

- `table` A table in the database.
- `database` A database.
- `owner` `nil`, `:all` or a string.

**Values**

- `result` A list of attributes.

**Description**

The function `list-attributes` returns a list of attributes from `table in database`, which has a default value of `*default-database*`. 
This chapter applies to the Enterprise Edition only

If `owner` is `nil`, only user-owned attributes are considered. This is the default.

If `owner` is `:all`, all attributes are considered.

If `owner` is a string, this denotes a username and only attributes owned by `owner` are considered.

See also
- `attribute-type`
- `list-attribute-types`
- `list-tables`

### list-classes

**Summary**
Returns a list of View Classes connected to a given database.

**Package**
sql

**Signature**
`list-classes &key database root-class test => result-list`

**Arguments**
- `database` A database.
- `root-class` A class.
- `test` A test function.

**Values**
- `result-list` A list of class objects.

**Description**
The function `list-classes` collects all the classes below `root-class` (which defaults to `standard-db-object`) that are connected to the given database specified by `database`, and which satisfy the `test` function. The default for the `test` argument is `cl:identity`.

By default, `list-classes` returns a list of all the classes connected to the default database, `*default-database*`. 
### list-sql-streams

**Function**

**Summary**
Returns the broadcast list of streams recording SQL commands or results traffic.

**Package**
sql

**Signature**
list-sql-streams &key type database => streams

**Arguments**
- **type** A keyword.
- **database** A database.

**Values**
- **streams** A list.

**Description**
The function `list-sql-streams` returns the broadcast list of streams recording SQL commands or results traffic.

Each element of `streams` is a stream or the symbol `t`, denoting *standard-output*.

The keyword `type` is one of `:commands` or `:results`, and determines whether to return a list of streams for SQL commands or results traffic.

The default value of `type` is `:commands`. The default value for `database` is the value of *default-database*.

**See also**
- add-sql-stream
- delete-sql-stream
- sql-recording-p
- sql-stream
- start-sql-recording
- stop-sql-recording

### list-tables

**Function**

**Summary**
Returns a list of the table names in a database.
This chapter applies to the Enterprise Edition only

Package   sql

Signature  list-tables &key database owner => table-list

Arguments  database        A database.
            owner          nil, :all or a string.

Values     table-list      A list of table names.

Description The function list-tables returns the list of table names in database, which has a default value of *default-database*.

If owner is nil, only user-owned tables are considered. This is the default.

If owner is :all, all tables are considered.

If owner is a string, this denotes a username and only tables owned by owner are considered.

See also create-table
        drop-table
        list-attributes
        table-exists-p

lob-stream  

Class

Summary    The LOB stream class.

Superclasses buffer-stream

Initargs   :lob-locator
            A LOB locator.

            :direction       One of :input or :output.
            :free-lob-locator-on-close
            A generalized boolean.
Accessors  lob-stream-lob-locator

Description  The class lob-stream implements LOB streams in the Oracle LOB interface.

A lob-stream for input can be returned from select or query by specifying :input-stream as the type to return for the LOB column.

A lob-stream for output can be returned from select or query by specifying :output-stream as the type to return for the LOB column.

A lob-stream can be attached to an existing LOB locator by creating the stream explicitly.

direction specifies whether the stream is for input or output. The default value of direction is :input.

By default, if the stream is closed the LOB locator is freed, unless free-lob-locator-on-close is passed as nil. The default value of free-lob-locator-on-close is t.

Example  This creates an input stream connected to the LOB locator lob-locator:

(make-instance 'lob-stream :lob-locator lob-locator)

See also  query
select

locally-disable-sql-reader-syntax

Function

Summary  Turns off square bracket syntax and does not change syntax state.

Package  sql

Signature  locally-disable-sql-reader-syntax =>
This chapter applies to the Enterprise Edition only

Arguments
None.

Values
None.

Description
The function `locally-disable-sql-reader-syntax` turns off square bracket syntax and does not change syntax state. This ensures that `restore-sql-reader-syntax-state` restores the current enable/disable state.

Example
The intended use of `locally-disable-sql-reader-syntax` is in a file:

```
#. (locally-disable-sql-reader-syntax)
<Lisp code not using [...] syntax>
#. (restore-sql-reader-syntax-state)
```

See also
`disable-sql-reader-syntax`
`enable-sql-reader-syntax`
`locally-enable-sql-reader-syntax`
`restore-sql-reader-syntax-state`

---

**locally-enable-sql-reader-syntax**

*Function*

Summary
Turns on square bracket syntax and does not change syntax state.

Package
`sql`

Signature
`locally-enable-sql-reader-syntax`

Arguments
None.

Values
None.

Description
The function `locally-enable-sql-reader-syntax` turns on square bracket syntax and does not change the syntax.
state. This ensures that `restore-sql-reader-syntax-state` restores the current enable/disable state.

Example
The intended use of `locally-enable-sql-reader-syntax` is in a file:

```
#. (locally-enable-sql-reader-syntax)
<code using [...] syntax>
#. (restore-sql-reader-syntax-state)
```

See also
`disable-sql-reader-syntax`
`enable-sql-reader-syntax`
`locally-disable-sql-reader-syntax`
`restore-sql-reader-syntax-state`

**loop**

Macro

Summary
Extends `loop` to provide a clause for iterating over query results.

Package
`common-lisp`

Signature
```
loop {for|as} var [type-spec] being {the|each} {records|record} {in|of} query-expression [not-inside-transaction not-inside-transaction] {get-all get-all} => result
```

Arguments
```
var A variable.
query-expression A SQL query statement or a `prepared-statement` containing a query.
not-inside-transaction A generalized boolean.
get-all A generalized boolean.
```

Values
```
result A `loop` return value.
```
This chapter applies to the Enterprise Edition only

**Description**

The Common Lisp `loop` macro has been extended with a clause for iterating over query results. This extension is available only when Common SQL has been loaded. For a full description of the rest of the Common Lisp `loop` facility, including the possible return values, see the ANSI Common Lisp specification.

Each iteration of the loop assigns the next record of the table to the variable `var`. The record is represented in Lisp as a list. Destructuring can be used in `var` to bind variables to specific attributes of the records resulting from `query-expression`. In conjunction with the panoply of existing clauses available from the `loop` macro, the new iteration clause provides an integrated report generation facility.

The additional loop keywords `not-inside-transaction` and `get-all` may be useful when fetching many records through a connection with `database-type` `mysql`. See the section “Special considerations for iteration functions and macros” on page 375 for details.

**Example**

This extended `loop` example performs the following on each record returned as a result of a query: bind `name` to the query result, find the salary (if any) from an associated hash-table, increment a count for salaries greater than 20000, accumulate the salary, and print the details. Finally, it prints the average salary.

```lisp
(loop
  for (name) being each record in
  [select [ename] :from [emp]]
  as salary = (gethash name "salary-table")
  initially (format t ""-&-20A-10D" 'name 'salary)
  when (and salary (> salary 20000))
    count salary into salaries
    and sum salary into total
    and do (format t ""-&-20A-10D" name salary)
  else
    do (format t ""-&-20A-10A" name "N/A")
  finally
    (format t ""-2&Av Salary: -10D" (/ total salaries)))
```
See also  
- do-query
- map-query
- prepare-statement
- query
- select
- simple-do-query

**map-query**

*Function*

**Summary**

Returns the results of mapping a function across a SQL query statement.

**Package**

sql

**Signature**

```lisp
map-query output-type-spec function query-exp &key database not-inside-transaction get-all => result
```

**Arguments**

- `output-type-spec`: The output type specification.
- `function`: A function.
- `query-exp`: A SQL query or a `prepared-statement` containing a query.
- `database`: A database.
- `get-all`: A generalized boolean.

**Values**

- `result`: A sequence of type `output-type-spec` containing the results of the map function.

**Description**

The function `map-query` returns the result of mapping `function` across the results of `query-exp`. The `output-type-spec` argument specifies the type of the result sequence as per the Common Lisp `map` function.
The default value of **database** is *default-database*.

*not-inside-transaction* and *get-all* may be useful when fetching many records through a connection with **database-type** :mysql. Both of these arguments have default value *nil*. See the section “Special considerations for iteration functions and macros” on page 375 for details.

**Example**

This example binds **name** to each name in the employee table and prints it.

```lisp
(map-query
 nil
 #'(lambda (name) (print name))
 [select [ename] :from [emp] :flatp t])
```

**See also**

*do-query*

*loop*

*prepare-statement*

*print-query*

*query*

*select*

*simple-do-query*

---

**mysql-library-directories**

**Variable**

**Package**

*sql*

**Initial value**

*nil*

**Description**

The variable *mysql-library-directories* helps Lisp-Works for Windows to locate the MySQL library for use with **database-type** :mysql.

It specifies a directory or a list of directories in which to search for the MySQL library. If the value is a directory pathname designator then it is passed to **directory**. If the value is a list of directory pathname designators then each item is
passed to directory. The collected results are the list of directories to search in.

**Notes**

The default value `nil` causes the system to use `*mysql-library-sub-directories*` to construct the search path. With the default installation of MySQL this copes better with 64-bit/32-bit mixing on the same machine. When `*mysql-library-directories*` is non-nil, it overrides `*mysql-library-sub-directories*`.

**Compatibility notes**

In LispWorks 6.0 `*mysql-library-directories*` has initial value `"C:\Program Files\MySQL\MySQL\bin"`. In LispWorks 6.1 and later, `*mysql-library-directories*` has initial value `nil` so the search path is constructed using `*mysql-library-sub-directories*`.

**See also**

`*mysql-library-path*`

`*mysql-library-sub-directories*`

---

**`*mysql-library-path*`**

**Variable**

<table>
<thead>
<tr>
<th>Package</th>
<th>sql</th>
</tr>
</thead>
</table>

**Initial value**

- On Microsoft Windows:
  
  `"libmysql.dll"`

- On other platforms with pthreads:
  
  `"-lmysqlclient_r"

- On other platforms without pthreads:
  
  `"-lmysqlclient"

**Description**

The variable `*mysql-library-path*` helps the system to locate the MySQL library for use with `database-type :mysql`. It specifies the library name, and can also be set to a full path. If
it is not a name, the system searches the standard library locations.

You can override the value of \*mysql-library-path\* by setting the environment variable LW_MYSQL_LIBRARY.

See also \*mysql-library-directories\* 

\*mysql-library-sub-directories\* Variable

Package sql

Initial value "MySQL\MySQL*\bin"

Description The variable \*mysql-library-sub-directories\* helps LispWorks for Windows to locate the MySQL library for use with \database-type mysql\.

It specifies a directory in which to search for the MySQL library, as a sub-directory of the appropriate Program Files directory. On a 32-bit machine that normally means C:\Program Files\, while on a 64-bit machine it normally means C:\Program Files for 64-bit programs and C:\Program Files (x86) for 32-bit programs.

The value must be a pathname designator. It is merged with the Program Files directory yielding a path (for example "C:\\Program Files\\MySQL\\MySQL*\\bin") which is then passed to \directory\.

The result is a list of directories that are used to search for the MySQL library.

The default value matches the default MySQL installation.

If \*mysql-library-directories\* is non-nil, it overrides \*mysql-library-sub-directories\*.

Note that this default will match any MySQL release, so if you need to be sure to match a specific MySQL release, you
need to change the value of *mysql-library-sub-directories* such that it matches only that particular release.

See also *mysql-library-directories*

**ora-lob-append**

*Function*

**Summary**

Appends two internal LOBs together.

**Package**

sql

**Signature**

`ora-lob-append src-lob-locator dest-lob-locator &key errorp`

**Arguments**

- `src-lob-locator` A LOB locator.
- `dest-lob-locator` A LOB locator.
- `errorp` A generalized boolean.

**Description**

The function `ora-lob-append` appends the contents of the LOB pointed to by `src-lob-locator` to the end of LOB pointed by `dest-lob-locator`. The source and destination LOBs must be of the same internal LOB type, that is, either both BLOB or both CLOB/NCLOB.

If an error occurs and `errorp` is true, an error is signaled. If `errorp` is false, the function returns an object of type `sql-database-error`. The default value of `errorp` is `nil`.

`ora-lob-append` is applicable to internal LOBs only.

**Notes**

1. `ora-lob-append` is a direct call OCILobAppend.
2. `ora-lob-append` is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.
ora-lob-assign

Function

Summary
Assigns a LOB to another LOB locator.

Package sql

Signature ora-lob-assign src-lob-locator &key dest-lob-locator errorp => lob-locator

Arguments
src-lob-locator A LOB locator.
dest-lob-locator A LOB locator.
errorp A generalized boolean.

Values lob-locator A LOB locator.

Description
The function ora-lob-assign assigns the underlying LOB for src-lob-locator to another LOB locator.

If dest-lob-locator is nil then a new LOB locator is created and returned. Otherwise dest-lob-locator should be an existing LOB locator which is modified and returned. The default value of dest-lob-locator is nil.

If an error occurs and errorp is true, an error is signaled. If errorp is false, the function returns an object of type sql-database-error. The default value of errorp is nil.

Notes
1. ora-lob-assign is a direct call to OCILOBAssign.
2. ora-lob-assign is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

ora-lob-char-set-form

Function

Summary
Returns the character set form of a LOB.
The SQL Package

This chapter applies to the Enterprise Edition only

Package sql

Signature ora-lob-char-set-form lob-locator &key errorp => charset

Arguments lob-locator A LOB locator.
errorp A generalized boolean.

Values charset A non-negative integer.

Description The function ora-lob-char-set-form returns the char set form of the LOB underlying lob-locator.

charset is 0 for a binary LOB (BLOB or BFILE), SQLCS_IMPLICIT (1) for a character LOB (CFILE or CLOB) and SQLCS_NCHAR (2) for a NCLOB.

If an error occurs and errorp is true, an error is signaled. If errorp is false, the function returns an object of type sql-database-error. The default value of errorp is nil.

Notes

1. This is a direct call to OCILobCharSetForm.
2. ora-lob-char-set-form is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

ora-lob-char-set-id Function

Summary Returns the database character set identifier of a LOB.

Package sql

Signature ora-lob-char-set-id lob-locator &key errorp => db-charset-id

Arguments lob-locator A LOB locator.
errorp A generalized boolean.
This chapter applies to the Enterprise Edition only

Values

| db-charset-id | A non-negative number. |

Description

The function `ora-lob-char-set-id` returns the database character set identifier of the LOB underlying `lob-locator`.

`db-charset-id` is 0 for a binary LOB.

If an error occurs and `errorp` is true, an error is signaled. If `errorp` is false, the function returns an object of type `sql-database-error`. The default value of `errorp` is `nil`.

Notes

1. This is a direct call to OCILobCharSetID.

2. `ora-lob-char-set-id` is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

---

**ora-lob-close**

*Function*

Summary

Closes an opened LOB.

Package

sql

Signature

`ora-lob-close lob-locator &key errorp`

Arguments

`lob-locator` A LOB locator.

`errorp` A generalized boolean.

Description

The function `ora-lob-close` closes a LOB which has been opened by `ora-lob-open`.

For more information see `ora-lob-open`.

If an error occurs and `errorp` is true, an error is signaled. If `errorp` is false, the function returns an object of type `sql-database-error`. The default value of `errorp` is `nil`.

Notes

1. This is a direct call to OCILobClose.
2. **ora-lob-close** is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

See also **ora-lob-open**

### ora-lob-copy

**Function**

**Summary**
Copies part of an internal LOB.

**Package**
sql

**Signature**
ora-lob-copy dest-lob-locator src-lob-locator amount &key dest-offset src-offset errorp

**Arguments**
- dest-lob-locator: A LOB locator.
- src-lob-locator: A LOB locator.
- amount: A non-negative integer.
- dest-offset: A non-negative integer.
- src-offset: A non-negative integer.
- errorp: A generalized boolean.

**Description**
The function **ora-lob-copy** copies part of the LOB pointed to by src-lob-locator into the LOB pointed to by dest-lob-locator. The details of the operation are determined by amount, src-offset and dest-offset. These numbers are in characters for CLOB/NCLOB and bytes for BLOB, and the offsets start from 1. The part of the source LOB from offset src-offset of length amount is copied into the destination LOB at offset dest-offset. The default value of dest-offset is 1 and the default value of src-offset is 1.

The destination LOB is extended if needed. If the dest-offset is beyond the end of the destination LOB, the gap between the
end and \textit{dest-offset} is erased, that is, filled with 0 for BLOBs or spaces for CLOBs.

Both LOBs must be internal LOBs, and they must be of the same type, that is, either both BLOB or both CLOB/NCLOB. \texttt{ora-lob-append} is applicable to internal LOBs only.

If an error occurs and \texttt{errorp} is true, an error is signaled. If \texttt{errorp} is false, the function returns an object of type \texttt{sql-database-error}. The default value of \texttt{errorp} is \texttt{nil}.

Notes
1. This is a direct call OCILobCopy.
2. This function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

See also \texttt{ora-lob-load-from-file}

\textbf{ora-lob-create-empty} \hspace{10cm} \textit{Function}

\textbf{Summary} \hspace{1cm} Creates an empty LOB.

\textbf{Package} \hspace{1cm} \texttt{sql}

\textbf{Signature} \hspace{1cm} \texttt{ora-lob-create-empty \&key db type \Rightarrow lob--locator}

\textbf{Arguments} \hspace{1cm} 
\begin{itemize}
  \item \texttt{db} \hspace{1cm} A database.
  \item \texttt{type} \hspace{1cm} A Lisp object.
\end{itemize}

\textbf{Values} \hspace{1cm} 
\begin{itemize}
  \item \texttt{lob-locator} \hspace{1cm} A LOB locator.
\end{itemize}

\textbf{Description} \hspace{1cm} The function \texttt{ora-lob-create-empty} creates an empty LOB object and returns a LOB locator for it.
If `type` is `:lob` then `ora-lob-create-empty` creates a LOB of type BLOB/CLOB. If `type` is any other value, it creates a file LOB. The default value of `type` is `:lob`.

Empty LOBs can be put in the database by passing them to `insert-records` or `update-records`. However, the preferred approach is to use the Oracle SQL function `EMPTY_BLOB` as described in the section “Inserting empty LOBs” on page 380.

The default value of `db` is the value of `*default-database*`.

**Notes**

`ora-lob-create-empty` is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

### `ora-lob-create-temporary`

**Function**

**Summary**

Creates a temporary LOB.

**Package**

`sql`

**Signature**

`ora-lob-create-temporary db-or-lob-locator &key errorp cache session-duration clob-p => lob-locator`

**Arguments**

- `db-or-lob-locator` A database or a LOB locator.
- `errorp` A generalized boolean.
- `cache` A generalized boolean.
- `session-duration` A generalized boolean.
- `clob-p` A generalized boolean.

**Values**

- `lob-locator` A LOB locator.
This chapter applies to the Enterprise Edition only

Description

The function `ora-lob-create-temporary` creates a temporary LOB.

`db-or-lob-locator` specifies the database to associate the new LOB with. If it is a LOB locator the database from which the LOB locator came is used.

If an error occurs and `errorp` is true, an error is signaled. If `errorp` is false, the function returns an object of type `sql-database-error`. The default value of `errorp` is `nil`.

`cache` specifies whether to use a cache or not. The default value of `cache` is `nil`.

`session-duration` specifies the lifetime: if it is true then it uses `OCI_DURATION_SESSION`, otherwise it uses `OCI_DURATION_CALL`. The default value of `session-duration` is `t`.

If `clob-p` is true then the new LOB is a CLOB, otherwise it is a BLOB. The default value of `clob-p` is `nil`.

The new temporary LOB locator is returned.

Notes

1. This is a direct call to OCILobCreateTemporary.

2. `ora-lob-create-temporary` is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

See also

`ora-lob-free-temporary`

`ora-lob-is-temporary`

ora-lob-disable-buffering

Function

Summary

Disables the buffering of the Oracle client.

Package

`sql`

Signature

`ora-lob-disable-buffering lob-locator &key errorp`
Arguments

<table>
<thead>
<tr>
<th>lob-locator</th>
<th>A LOB locator.</th>
</tr>
</thead>
<tbody>
<tr>
<td>errorp</td>
<td>A generalized boolean.</td>
</tr>
</tbody>
</table>

Description

The function `ora-lob-disable-buffering` disables the buffering of the Oracle client. This function does not flush the buffers.

This function is applicable to internal LOBs only.

If an error occurs and `errorp` is true, an error is signaled. If `errorp` is false, the function returns an object of type `sql-database-error`. The default value of `errorp` is `nil`.

Notes

1. This is a direct call to OCILobDisableBuffering.
2. `ora-lob-disable-buffering` is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

See also

- `ora-lob-enable-buffering`
- `ora-lob-flush-buffer`

### ora-lob-element-type

**Function**

**Summary**

Returns the Lisp element type corresponding to that of a LOB locator.

**Package**

`sql`

**Signature**

`ora-lob-element-type lob-locator => type`

**Arguments**

| lob-locator | A LOB locator. |

**Values**

| type | A Lisp type descriptor. |
This chapter applies to the Enterprise Edition only

| Description | The function **ora-lob-element-type** returns the Lisp element type that best corresponds to the charset of the LOB locator **lob-locator**. For BLOB and BFILE type is **(unsigned-byte 8)**. For CLOB, NCLOB and CFILE type is either **base-char** or **simple-char**, depending on the charset. |
| Notes | **ora-lob-element-type** is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information. |

### ora-lob-enable-buffering

**Function**

| Summary | Enables the buffering of the Oracle client. |
| Package | sql |
| Signature | **ora-lob-enable-buffering** lob-locator &key errorp |
| Arguments | lob-locator A LOB locator. errorp A generalized boolean. |
| Description | The function **ora-lob-enable-buffering** enables the buffering of the Oracle client. This function does not flush the buffers. This function is applicable to internal LOBs only. If an error occurs and **errorp** is true, an error is signaled. If **errorp** is false, the function returns an object of type **sql-database-error**. The default value of **errorp** is **nil**. |
| Notes | 1. This is a direct call to OCILobEnableBuffering. 2. **ora-lob-enable-buffering** is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information. |
See also  
ora-lob-disable-buffering
ora-lob-flush-buffer

**ora-lob-env-handle**

*Function*

**Summary**
Returns a foreign pointer to the environment handle of a LOB.

**Package**
sql

**Signature**
ora-lob-env-handle  lob-locator  =>  pointer

**Arguments**
lob-locator  A LOB locator.

**Values**
pointer  A foreign pointer of type p oci env.

**Description**
The function ora-lob-env-handle returns a foreign pointer to the environment handle of the LOB underlying lob-locator.

**Notes**
ora-lob-env-handle is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

**ora-lob-erase**

*Function*

**Summary**
Erases part of an internal LOB.

**Package**
sql

**Signature**
ora-lob-erase  lob-locator  offset  amount  &key  errorp  =>  erased

**Arguments**
lob-locator  A LOB locator.
offset  A non-negative integer.
amount  A non-negative integer.
This chapter applies to the Enterprise Edition only

errorp  A generalized boolean.

Values  erased  A non-negative integer.

Description  The function *ora-lob-erase* erases part of the LOB pointed to by *lob-locator*. That is, it fills part of the LOB with 0 for BLOBs or spaces for CLOBs.

The operation starts from offset *offset* into the LOB and erases *amount* of data in the LOB, or to the end of the LOB. Note that the offset starts from 1, and that *offset* and *amount* are in characters for CLOBs and bytes for BLOB.

Erasing does not extend beyond the end of the LOB. The return value *erased* is the number of characters or bytes erased. *erased* will be smaller than *amount* if the sum of *offset* and *amount* is greater than the length of the LOB.

*ora-lob-erase* is applicable to internal LOBs only.

If an error occurs and *errorp* is true, an error is signaled. If *errorp* is false, the function returns an object of type *sql-database-error*. The default value of *errorp* is *nil*.

Notes
1. This is a direct call to OCILobErase.
2. *ora-lob-erase* is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

**ora-lob-file-close**

*Summary*  Closes a file LOB.

*Package*  sql

*Signature*  ora-lob-file-close file-lob-locator &key errorp

*Arguments*  file-lob-locator  A file LOB locator.
**errorp**

A generalized boolean.

**Description**

The function `ora-lob-file-close` closes the file that `file-lob-locator` is associated with.

If an error occurs and `errorp` is true, an error is signaled. If `errorp` is false, the function returns an object of type `sql-database-error`. The default value of `errorp` is `nil`.

**Notes**

1. This is a direct call to OCILobFileClose.
2. `ora-lob-file-close` is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

**See also**

`ora-lob-file-open`

---

**ora-lob-file-close-all**

**Function**

**Summary**

Closes all the file LOBs.

**Package**

`sql`

**Signature**

`ora-lob-file-close-all &key db errorp`

**Arguments**

`db`  
A database.

`errorp`  
A generalized boolean.

**Description**

The function `ora-lob-file-close-all` closes the files that are associated with all the file LOB locators that are opened through the database connection specified by `database`.

The default value of `db` is the value of `*default-database*`.

If an error occurs and `errorp` is true, an error is signaled. If `errorp` is false, the function returns an object of type `sql-database-error`. The default value of `errorp` is `nil`. 
This chapter applies to the Enterprise Edition only

Notes

1. This is a direct call to OCILobFileCloseAll.

2. ora-lob-file-close-all is available only when the “oracle” module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

See also ora-lob-file-close

ora-lob-file-exists

Function

Summary

The predicate for whether a LOB file exists.

Package

sql

Signature

ora-lob-file-exists lob-locator &key errorp => result

Arguments

lob-locator A LOB locator.

errorp A generalized boolean.

Values

result A boolean.

Description

The function ora-lob-file-exists returns \texttt{t} if the file associated with the LOB exists. This function is applicable only to file LOBs (CFILE or BFILE).

If an error occurs and \texttt{errorp} is true, an error is signaled. If \texttt{errorp} is false, the function returns an object of type \texttt{sql-database-error}. The default value of \texttt{errorp} is \texttt{nil}.

Notes

1. This is a direct call to OCILobFileExists.

2. ora-lob-file-exists is available only when the “oracle” module is loaded. See the section “Oracle LOB interface” on page 380 for more information.
ora-lob-file-get-name

**Summary**
Returns the directory and name for the file associated with a file LOB.

**Package**
`sql`

**Signature**
`ora-lob-file-get-name lob-locator &key errorp => dir, filename`

**Arguments**
- `lob-locator` A LOB locator.
- `errorp` A generalized boolean.

**Values**
- `dir` A string of length no greater than 30.
- `filename` A string of length no greater than 255.

**Description**
The function `ora-lob-file-get-name` returns as multiple values the directory alias `dir` and the filename `filename` associated with the LOB denoted by `lob-locator`. The function is applicable only to file LOBs (CFILE or BFILE).

If an error occurs and `errorp` is true, an error is signalled. If `errorp` is false, the function returns an object of type `sql-database-error`. The default value of `errorp` is `nil`.

**Notes**
1. This is a direct call to OCILobFileGetName.
2. `ora-lob-file-get-name` is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

ora-lob-file-is-open

**Summary**
The predicate for whether a LOB file is open.

**Package**
`sql`
This chapter applies to the Enterprise Edition only

**Signature**

ora-lob-file-is-open lob-locator &key errorp => result

**Arguments**

- lob-locator: A LOB locator.
- errorp: A generalized boolean.

**Values**

- result: A boolean.

**Description**

The function **ora-lob-file-is-open** returns t if the file associated with the LOB is open. This function is applicable only to file LOBs (CFILE or BFILE).

If an error occurs and errorp is true, an error is signaled. If errorp is false, the function returns an object of type sql-database-error. The default value of errorp is nil.

**Notes**

1. This is a direct call to OCILobFileIsOpen.
2. **ora-lob-file-is-open** is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

**ora-lob-file-open**

**Function**

**Summary**

Opens a file LOB.

**Package**

sql

**Signature**

ora-lob-file-open file-lob-locator &key errorp

**Arguments**

- file-lob-locator: A file LOB locator.
- errorp: A generalized boolean.

**Description**

The function **ora-lob-file-open** opens the file that file-lob-locator is associated with.
If an error occurs and errorp is true, an error is signaled. If errorp is false, the function returns an object of type sql-database-error. The default value of errorp is nil.

Notes
1. This is a direct call to OCILobFileOpen.
2. ora-lob-file-open is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

See also    ora-lob-file-close

ora-lob-file-set-name

Summary    Sets the name of a file LOB.

Package    sql

Signature   ora-lob-file-set-name file-lob-locator dir-alias name &key errorp

Arguments
file-lob-locator    A file LOB locator.

dir-alias          A string or nil.

name               A string or nil.

errorp             A generalized boolean.

Description The function ora-lob-file-set-name sets the directory alias and the name of the file LOB pointed to by file-lob-locator.

If dir-alias is a string it should be of length no greater than 30. If it is nil then the directory alias of the file LOB is not changed.

If name is a string it should be of length no greater than 255. If it is nil then the name of the file LOB is not changed.
If an error occurs and errorp is true, an error is signaled. If errorp is false, the function returns an object of type sql-database-error. The default value of errorp is nil.

Notes

1. This is a direct call to OCILobFileSetAlias.
2. ora-lob-file-set-name is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

ora-lob-flush-buffer

Function

Summary
Flushes the buffer of the Oracle client.

Package
sql

Signature
ora-lob-flush-buffer lob-locator &key free-buffer errorp

Arguments

lob-locator A LOB locator.
free-buffer A generalized boolean.
errorp A generalized boolean.

Description
The function ora-lob-flush-buffer flushes the buffer that is used by the Oracle client.

If free-buffer is true, it also frees the buffer. The default value of free-buffer is nil.

If an error occurs and errorp is true, an error is signaled. If errorp is false, the function returns an object of type sql-database-error. The default value of errorp is nil.

Notes

1. This is a direct call to OCILobFlushBuffer.
2. ora-lob-flush-buffer is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.
ora-lob-free

**Function**

Frees a LOB locator.

**Summary**

Frees a LOB locator.

**Package**

sql

**Signature**

ora-lob-free  lob-locator

**Arguments**

lob-locator A LOB locator.

**Description**

The function ora-lob-free frees the LOB locator lob-locator. If lob-locator was retrieved inside an iteration macro or function (that is, one of map-query, do-query, simple-do-query and loop), it is freed before the next record is fetched, or when terminating the iteration for the last record.

LOB locators which were retrieved by select or query, or were created by the user by ora-lob-assign or ora-lob-create-empty are freed automatically when the database connection is closed by a call to disconnect.

If you create many LOB locators without closing the connection, it is useful to free them by calling ora-lob-free, to free the resources that are associated with them.

Freeing a LOB locator does not affect the underlying LOB. In particular, after modifications to the LOB there is no rollback even if there was not yet a commit.

**Notes**

ora-lob-free is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.
ora-lob-free-temporary

**Function**

**Summary**
Frees a temporary LOB locator.

**Package**
sql

**Signature**
ora-lob-free-temporary temp-lob-locator &key errorp

**Arguments**
temp-lob-locator  A temporary LOB locator.

errorp  A generalized boolean.

**Description**
The function *ora-lob-free-temporary* frees a temporary LOB locator.

`temp-lob-locator` should be a temporary LOB locator as created by *ora-lob-create-temporary*.

If an error occurs and `errorp` is true, an error is signaled. If `errorp` is false, the function returns an object of type *sql-database-error*. The default value of `errorp` is `nil`.

**Notes**
1. Temporary LOB locators are freed automatically when the database connection is closed by `disconnect`.
2. This is a direct call to `OCILobFreeTemporary`.
3. *ora-lob-free-temporary* is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

**See also**
ora-lob-create-temporary
ora-lob-is-temporary

ora-lob-get-buffer

**Function**

**Summary**
 Gets a buffer for efficient I/O with foreign functions.

**Package**
sql
The function *ora-lob-get-buffer* gets a buffer for efficient I/O with foreign functions.

If `for-writing` is `nil`, then `ora-lob-get-buffer` fills an internal buffer and returns three values: `amount/size` is how much it filled, `foreign-buffer` points to the actual buffer, and `eof-or-error-p` is the return value from the call to `ora-lob-read-foreign-buffer`. The offset `offset` is passed directly `ora-lob-read-foreign-buffer`.

If `for-writing` is true, then `ora-lob-get-buffer` returns two values: `amount/size` is the size of the foreign buffer and `foreign-buffer` points to the actual buffer, which then can be passed to `ora-lob-write-foreign-buffer`.

The default value of `for-writing` is `nil`.

The buffer that is used by `ora-lob-get-buffer` is always the same for the LOB locator, it is used by `ora-lob-read-buffer` and `ora-lob-write-buffer`, and is freed automatically when the LOB locator is freed. Thus until you finish with the buffer, you cannot use `ora-lob-read-buffer` or `ora-lob-write-buffer` or call `ora-lob-get-buffer` again or free the LOB locator.
This chapter applies to the Enterprise Edition only

Notes  
ora-lob-get-buffer is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

Example  
This first example illustrates reading using the buffer obtained by ora-lob-get-buffer. You have a foreign function:

my_chunk_processor(char *data, int size)

with this FLI definition:

(fli:define-foreign-function my_chunk_processor
 ((data :pointer)
  (size :int)))

You can pass all the data from the LOB locator to this function. Assuming no other function reads from it, it will start from the beginning.

(loop
 (multiple-value-bind (amount buffer eof-or-error-p)
    (ora-lob-get-buffer lob)
    (when (zerop amount) (return))
    (my_chunk_processor buffer amount ))

This second example illustrates writing with the buffer obtained by ora-lob-get-buffer. You have a foreign function that fills a buffer with data, and you want to write it to a LOB. First you should lock the record, and if required trim the LOB locator.

(multiple-value-bind (size buffer)
    (ora-lob-get-buffer lob-locator
     :for-writing t
     ;; start at the beginning
     :offset 1)
    (loop (let ((amount (my-fill-buffer buffer size)))
       (when (zerop amount) (return))
       (ora-lob-write-foreign-buffer
        lob-locator nil
        amount buffer size))))
See also
ora-lob-read-buffer
ora-lob-read-foreign-buffer
ora-lob-write-buffer
ora-lob-write-foreign-buffer

ora-lob-get-chunk-size  

Function

Summary
Returns the chunk size of a LOB.

Package
sql

Signature
ora-lob-get-chunk-size lob-locator &key errorp => size

Arguments
lob-locator A LOB locator.
errorp A generalized boolean.

Values
size A non-negative integer.

Description
The function ora-lob-get-chunk-size returns the chunk size of the LOB locator lob-locator, which is the best value for the size of a buffer.

If an error occurs and errorp is true, an error is signaled. If errorp is false, the function returns an object of type sql-database-error. The default value of errorp is nil.

Notes
1. This is a direct call to OCILobGetChunkSize.
2. ora-lob-get-chunk-size is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

ora-lob-get-length  

Function

Summary
Returns the length of a LOB.
This chapter applies to the Enterprise Edition only

Package  
sql

Signature  
ora-lob-get-length  lob-locator &key errorp => length

Arguments  
lob-locator  A LOB locator.
errorp  A generalized boolean.

Values  
length  A non-negative integer.

Description  
The function ora-lob-get-length returns the current length of the LOB underlying lob-locator.

If an error occurs and errorp is true, an error is signaled. If errorp is false, the function returns an object of type sql-database-error. The default value of errorp is nil.

Notes  
1. This is a direct call to OCILobGetLength.
2. ora-lob-get-length is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

ora-lob-internal-lob-p

Function

Summary  
The predicate for internal LOBs.

Package  
sql

Signature  
ora-lob-internal-lob-p  lob-locator => result

Arguments  
lob-locator  A LOB locator.

Values  
result  A boolean.

Description  
The function ora-lob-internal-lob-p returns t if lob-locator is internal (BLOB, CLOB, or NCLOB). Otherwise it returns nil.
**ora-lob-is-equal**

**Function**

**Summary**
The comparison function for LOB locators.

**Package**
sql

**Signature**
ora-lob-is-equal lob-locator1 lob-locator2 => result

**Arguments**
- lob-locator1: A LOB locator.
- lob-locator2: A LOB locator.

**Values**
- result: A boolean.

**Description**
The function ora-lob-is-equal returns t if lob-locator1 and lob-locator2 point to the same LOB object.

**Notes**

1. This is a direct call to OCILobIsEqual.
2. ora-lob-is-equal is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

---

**ora-lob-is-open**

**Function**

**Summary**
The predicate for whether a LOB locator is opened.

**Package**
sql

**Signature**
ora-lob-is-open lob-locator &key errorp => result

**Arguments**
- lob-locator: A LOB locator.
This chapter applies to the Enterprise Edition only

errorp  A generalized boolean.

<table>
<thead>
<tr>
<th>Values</th>
<th>result</th>
<th>A boolean.</th>
</tr>
</thead>
</table>

Description

The function *ora-lob-is-open* returns t if the LOB pointed to by *lob-locator* is opened (by *ora-lob-open*).

*ora-lob-is-open* is applicable to internal LOBs only.

If an error occurs and *errorp* is true, an error is signaled. If *errorp* is false, the function returns an object of type *sql-database-error*. The default value of *errorp* is *nil*.

Notes

1. This is a direct call to OCILOBIsOpen.
2. *ora-lob-is-open* is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

See also

*ora-lob-open*

**Function**

**ora-lob-is-temporary**

Summary

The predicate for whether a LOB is temporary.

Package

*sql*

Signature

*ora-lob-is-temporary* *lob-locator* &key *errorp* => *result*

Arguments

- *lob-locator*  A LOB locator.
- *errorp*  A generalized boolean.

Values

- *result*  A boolean.

Description

The function *ora-lob-is-temporary* returns t if the LOB underlying *lob-locator* is temporary, that is, it was created by *ora-lob-create-temporary*.
If an error occurs and \textit{errorp} is true, an error is signaled. If \textit{errorp} is false, the function returns an object of type \texttt{sql-database-error}. The default value of \textit{errorp} is \texttt{nil}.

\textbf{Notes}

1. This is a direct call to OCILobIsTemporary.
2. \texttt{ora-lob-is-temporary} is available only when the “oracle” module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

\textbf{See also} \quad \texttt{ora-lob-create-temporary}

\section*{ora-lob-load-from-file}

\textbf{Function}

\textbf{Summary} \quad Loads data from a file LOB into a LOB.

\textbf{Package} \quad \texttt{sql}

\textbf{Signature} \quad \texttt{ora-lob-load-from-file dest-lob-locator src-lob-file amount \&key src-offset dest-offset errorp}

\textbf{Arguments} \quad \begin{itemize}
\item \texttt{dest-lob-locator} \quad An internal LOB locator.
\item \texttt{src-lob-file} \quad A file LOB locator.
\item \texttt{amount} \quad A non-negative integer.
\item \texttt{src-offset} \quad A non-negative integer.
\item \texttt{dest-offset} \quad A non-negative integer.
\item \texttt{errorp} \quad A generalized boolean.
\end{itemize}

\textbf{Description} \quad The function \texttt{ora-lob-load-from-file} loads the data from the \texttt{src-lob-file} into the destination LOB pointed to by \texttt{dest-lob-locator}.

The source LOB must be a BFILE and the destination must be an internal LOB.
This chapter applies to the Enterprise Edition only

The details of the operation are determined by amount, src-offset and dest-offset. amount and dest-offset are in characters for CLOB/NCLOB and are in bytes for BLOB. src-offset is in bytes. The offsets start from 1. The default value of dest-offset is 1 and the default value of src-offset is 1.

No conversion is performed by ora-lob-load-from-file, so if the destination is a CLOB/NCLOB, the source must already be in the right format.

If an error occurs and errorp is true, an error is signaled. If errorp is false, the function returns an object of type sql-database-error. The default value of errorp is nil.

Notes
1. This is a direct call to OCILobLoadFromFile. The Oracle documentation is ambiguous on whether it is mandatory to open the source LOB before calling this function.
2. ora-lob-load-from-file is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

See also ora-lob-copy

ora-lob-lob-locator

Function

Summary
Returns a foreign pointer to the underlying LOB locator.

Package
sql

Signature
ora-lob-lob-locator  lob-locator  =>  pointer

Arguments
lob-locator  A LOB locator.

Values
pointer  A foreign pointer.

Description
The function ora-lob-lob-locator returns a foreign pointer to the OCI LOB locator underlying lob-locator.
pointer is of type \texttt{p-oci-lob-locator} or \texttt{p-oci-file}.

Notes
\texttt{ora-lob-lob-locator} is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

\textbf{ora-lob-locator-is-init}

\textbf{Function}

\textbf{Summary}
The predicate for whether a LOB is initialized.

\textbf{Package}
\texttt{sql}

\textbf{Signature}
\texttt{ora-lob-locator-is-init lob-locator &key errorp => result}

\textbf{Arguments}
\begin{itemize}
  \item \texttt{lob-locator} A LOB locator.
  \item \texttt{errorp} A generalized boolean.
\end{itemize}

\textbf{Values}
\begin{itemize}
  \item \texttt{result} A boolean.
\end{itemize}

\textbf{Description}
The function \texttt{ora-lob-locator-is-init} returns \texttt{t} if the LOB locator \texttt{lob-locator} is initialized.

If an error occurs and \texttt{errorp} is true, an error is signaled. If \texttt{errorp} is false, the function returns an object of type \texttt{sql-database-error}. The default value of \texttt{errorp} is \texttt{nil}.

\textbf{Notes}
\begin{enumerate}
  \item This is a direct call to OCILobIsInit.
  \item \texttt{ora-lob-locator-is-init} is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.
\end{enumerate}

\textbf{ora-lob-open}

\textbf{Function}

\textbf{Summary}
Opens a LOB.
### Package

sql

### Signature

`ora-lob-open lob-locator &key errorp`

### Arguments

- **lob-locator**: A LOB locator.
- **errorp**: A generalized boolean.

### Description

The function `ora-lob-open` opens the LOB pointed to by `lob-locator`, which can be an internal LOB or a file LOB. Opening the LOB creates a transaction, so any updates associated with modifying the LOB are delayed until the `ora-lob-close` call. This saves round-trips and avoids extra work on the server side. However it is not mandatory to use `ora-lob-open`.

Calls to `ora-lob-open` must be strictly paired to calls to `ora-lob-close`, and the latter must be called before a call to `commit`. It is also an error to call `ora-lob-open` on a server LOB object that is already open, even if it has been opened via a different LOB locator.

If an error occurs and `errorp` is true, an error is signaled. If `errorp` is false, the function returns an object of type `sql-database-error`. The default value of `errorp` is `nil`.

### Notes

1. This is a direct call to OCILobOpen.
2. `ora-lob-open` is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

### See also

- `ora-lob-close`
- `ora-lob-is-open`

---

### Function

**ora-lob-read-buffer**

**Summary**: Reads from a LOB into a buffer.
### SQL Package

This chapter applies to the Enterprise Edition only.

<table>
<thead>
<tr>
<th>Package</th>
<th>sql</th>
</tr>
</thead>
</table>

| Signature | ora-lob-read-buffer lob-locator offset amount buffer &key buffer-offset csid => amount-read, eof-or-error-p |

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lob-locator</td>
<td>A LOB locator.</td>
</tr>
<tr>
<td>offset</td>
<td>A non-negative integer or nil.</td>
</tr>
<tr>
<td>amount</td>
<td>A non-negative integer.</td>
</tr>
<tr>
<td>buffer</td>
<td>A string, or a vector of element type (unsigned-byte 8).</td>
</tr>
<tr>
<td>buffer-offset</td>
<td>A non-negative integer.</td>
</tr>
<tr>
<td>csid</td>
<td>A Character Set ID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount-read</td>
<td>A non-negative integer.</td>
</tr>
<tr>
<td>eof-or-error-p</td>
<td>A boolean or an error object.</td>
</tr>
</tbody>
</table>

**Description**

The function `ora-lob-read-buffer` reads into `buffer` from the LOB pointed to by `lob-locator`.

`offset` specifies the offset to start reading from. It starts with 1, and specifies characters for CLOB/NCLOB/CFILE and bytes for BLOB/BFILE. If offset is nil then the offset after the end of the previous read operation is used (write operations are ignored). This is especially useful for reading linearly from the LOB.

`amount` is the amount to read, in characters for CLOB/NCLOB/CFILE and bytes for BLOB/BFILE.

The element type of `buffer` should match the element type of the LOB locator (see `ora-lob-element-type`). For this comparison (unsigned-byte 8) and base-char are considered as the same.

If the buffer `buffer` is not static, there is some additional overhead. For small amounts of data, this is probably insignificant.
buffer-offset specifies where to put the data. It is an offset in bytes from the beginning of the buffer. The default value of buffer-offset is 0.

csid specifies what Character Set ID the data in the target buffer should be. It defaults to the CSID of the LOB pointed to by lob-locator.

The return value amount-read is the number of elements (characters or bytes) that were read.

If the return value eof-or-error-p is nil then there is still more to read. If eof-or-error-p is t then it read to the end of the LOB. If an error occurred then eof-or-error-p is an error object.

Notes
1. This is a direct call to OCILobRead, without callback.
2. ora-lob-read-buffer is available only when the "oracle" module is loaded. See the section "Oracle LOB interface" on page 380 for more information.

Example
This example sequentially reads the LOB data into a string, starting from offset 10000. It calls a processing function on each chunk of data and then reads in the next chunk starting from where the previous read ended.

(let ((my-buffer (make-string 1000
:element-type 'base-char))
(offset 10000))
(loop
(let ((nread
(ora-lob-read-buffer lob-locator
offset
1000
my-buffer)))
(when (zerop nread) ; end of the LOB
(return))
(my-processing-function my-buffer nread))
(setq offset nil))) ; so next time it continues ; from where it finished

See also
ora-lob-element-type
ora-lob-read-foreign-buffer
ora-lob-read-into-plain-file  

Function

Summary  Writes the contents of a LOB into a file.

Package  sql

Signature  ora-lob-read-into-plain-file lob-locator file-name &key offset file-offset if-exists

Arguments  

lob-locator  A LOB locator.

file-name  A pathname designator.

offset  A non-negative integer, or nil.

file-offset  A non-negative integer, or nil.

if-exists  A keyword or nil.

Description  

The function ora-lob-read-into-plain-file writes the contents of a LOB into a file.

file-name specifies the file to write, which should be a standard file. The file is always opened in a binary mode, so if the LOB is a CLOB, the file will be generated in the right format when reading it from the LOB.

offset is the offset into the LOB from where to start reading. It starts from 1, counts characters in a CLOB, and if it is nil then the operation starts from the end of the previous read operation. The default value of offset is nil.

file-offset specifies the offset into the file to start the operation from. If file-offset is nil then it starts writing at the start of the file. The default value of file-offset is nil.

if-exists is passed to open when opening the file, with the standard Common Lisp meaning. The default value of if-exists is :error.
This chapter applies to the Enterprise Edition only

Notes

*ora-lob-read-into-plain-file* is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

See also

*ora-lob-write-from-plain-file*

---

**ora-lob-read-foreign-buffer**

**Function**

**Summary**

Reads from a LOB into a foreign buffer.

**Package**

sql

**Signature**

\[
\text{ora-lob-read-foreign-buffer} \quad \text{lob-locator} \quad \text{offset} \quad \text{amount} \quad \text{foreign-buffer} \quad \text{buffer-length} \quad \&\text{key} \quad \text{buffer-offset} \quad \text{csid} = \Rightarrow \text{amount-read}, \text{eof-or-error-p}
\]

**Arguments**

- **lob-locator**: A LOB locator.
- **offset**: A non-negative integer or **nil**.
- **amount**: A non-negative integer.
- **foreign-buffer**: A FLI pointer.
- **buffer-length**: A non-negative integer.
- **buffer-offset**: A non-negative integer.
- **csid**: A character set ID

**Values**

- **amount-read**: A non-negative integer.
- **eof-or-error-p**: A boolean or an error object.

**Description**

The function *ora-lob-read-foreign-buffer* reads from the LOB pointed to by **lob-locator** into the foreign buffer **foreign-buffer**.

This is just like *ora-lob-read-buffer* except that it reads from the LOB locator into a foreign buffer.
foreign-buffer is a FLI pointer to a buffer, which must be of size at least buffer-length.

Notes

1. This is a direct call to OCILobRead, without callback.

2. ora-lob-read-foreign-buffer is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

See also

ora-lob-get-buffer
ora-lob-read-buffer

ora-lob-svc-ctx-handle

Function

Summary

Returns a foreign pointer to the context handle of a LOB.

Package

sql

Signature

ora-lob-svc-ctx-handle lob-locator => pointer

Arguments

lob-locator A LOB locator.

Values

pointer A foreign pointer of type p-oci-svc-ctx.

Description

The function ora-lob-svc-ctx-handle returns a foreign pointer to the context handle of the LOB underlying lob-locator.

Notes

ora-lob-svc-ctx-handle is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

ora-lob-trim

Function

Summary

Trims an internal LOB.
This chapter applies to the Enterprise Edition only

**ora-lob-trim**

**Package**  
sql

**Signature**  
ora-lob-trim lob-locator new-size &key errorp

**Arguments**  
lob-locator A LOB locator.
new-size A non-negative integer.
errorp A generalized boolean.

**Description**  
The function *ora-lob-trim* trims the LOB pointed to by *lob-locator* to a new size *new-size*, which must be smaller than its current size.

Note that *new-size* is in characters for CLOBs and bytes for BLOBs.

*ora-lob-trim* is applicable to internal LOBs only.

If an error occurs and *errorp* is true, an error is signaled. If *errorp* is false, the function returns an object of type *sql-database-error*. The default value of *errorp* is *nil*.

**Notes**

1. This is a direct call to OCILobTrim.
2. *ora-lob-trim* is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

---

**ora-lob-write-buffer**

**Function**

**Summary**  
Writes a buffer to a LOB.

**Package**  
sql

**Signature**  
ora-lob-write-buffer lob-locator offset amount buffer &key
buffer-offset csid => amount-written, eof-or-error-p

**Arguments**  
lob-locator A LOB locator.
offset A non-negative integer or nil.
amount A non-negative integer.
buffer A string, or a vector of element type (unsigned-byte 8).
buffer-offset A non-negative integer.
csid A Character Set ID

Values

amount-written A non-negative integer.
eof-or-error-p A boolean or an error object.

Description

The function ora-lob-write-buffer writes to the LOB pointed to by lob-locator from buffer.

offset specifies the offset to start writing to. It starts with 1, and specifies characters for CLOB/NCLOB/CFILE and bytes for BLOB/BFILE. If offset is nil then the offset after the end of the previous write operation is used (read operations are ignored). This is especially useful for writing linearly to the LOB.

amount is the amount to write, in characters for CLOB/NCLOB/CFILE and bytes for BLOB/BFILE.

The element type of buffer should match the element type of the LOB locator (see ora-lob-element-type). For this comparison (unsigned-byte 8) and base-char are considered as the same.

If the buffer buffer is not static, there is some additional overhead. For small amounts of data, this is probably insignificant.

buffer-offset specifies where in the buffer to start writing data from. It is an offset in bytes from the beginning of the buffer. The default value of buffer-offset is 0.
This chapter applies to the Enterprise Edition only

`csid` specifies what Character Set ID the data in the source buffer should be. It defaults to the CSID of the LOB pointed to by `lob-locator`.

The return value `amount-written` is the number of elements (characters or bytes) that were written.

The LOB is extended as required.

If the return value `eof-or-error-p` is `nil` then there is still more to write. If `eof-or-error-p` is `t` then it wrote to the end of the LOB. If an error occurred then `eof-or-error-p` is an error object.

Notes
1. The record from which the LOB came must be locked. See the section “Locking” on page 382.
2. `ora-lob-write-buffer` is a direct call to OCILobWrite, without callback.
3. `ora-lob-write-buffer` is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

See also
- `ora-lob-element-type`
- `ora-lob-write-foreign-buffer`

**ora-lob-write-from-plain-file**

**Function**

**Summary**
Writes the contents of a file into a LOB.

**Package**
`sql`

**Signature**
`ora-lob-write-from-plain-file lob-locator file-name &key offset file-offset if-does-not-exist`

**Arguments**
- `lob-locator` A LOB locator.
- `file-name` A pathname designator.
- `offset` A non-negative integer, or `nil`.
file-offset  A non-negative integer, or nil.
if-does-not-exist A keyword or nil.

Description
The function ora-lob-write-from-plain-file writes the contents of a file into a LOB.

file-name specifies the file to read, which should be a standard file. The file is always opened in a binary mode, so if the LOB is a CLOB, the file must be in the right format when writing it into the LOB.

offset is the offset into the LOB from where to start writing. It starts from 1, counts characters in a CLOB, and if it is nil then the operation starts from the end of the previous write operation. The default value of offset is nil.

file-offset specifies the offset into the file to start the operation from. If file-offset is nil then it starts reading at the start of the file. The default value of file-offset is nil.

if-does-not-exist is passed to open when opening the file, with the standard Common Lisp meaning. The default value of if-does-not-exist is :error.

Notes
ora-lob-write-from-plain-file is available only when the “oracle” module is loaded. See the section “Oracle LOB interface” on page 380 for more information.

See also
ora-lob-read-into-plain-file

ora-lob-write-foreign-buffer

Function

Summary
Writes a foreign buffer to a LOB.

Package
sql
This chapter applies to the Enterprise Edition only

| Signature               | `ora-lob-write-foreign-buffer lob-locator offset amount
foreign-buffer buffer-length &key buffer-offset csid => amount-written, eof-or-error-p` |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td><code>lob-locator</code> A LOB locator.</td>
</tr>
<tr>
<td></td>
<td><code>offset</code> A non-negative integer or <code>nil</code>.</td>
</tr>
<tr>
<td></td>
<td><code>amount</code> A non-negative integer.</td>
</tr>
<tr>
<td></td>
<td><code>foreign-buffer</code> A FLI pointer.</td>
</tr>
<tr>
<td></td>
<td><code>buffer-length</code> A non-negative integer.</td>
</tr>
<tr>
<td></td>
<td><code>buffer-offset</code> A non-negative integer.</td>
</tr>
<tr>
<td></td>
<td><code>csid</code> A Character Set ID</td>
</tr>
<tr>
<td>Values</td>
<td><code>amount-written</code> A non-negative integer.</td>
</tr>
<tr>
<td></td>
<td><code>eof-or-error-p</code> A boolean or an error object.</td>
</tr>
<tr>
<td>Description</td>
<td>The function <code>ora-lob-write-foreign-buffer</code> writes to the LOB pointed to by <code>lob-locator</code> from <code>buffer</code>.</td>
</tr>
<tr>
<td></td>
<td>This is just like <code>ora-lob-write-buffer</code> except that it writes the LOB locator from a foreign buffer.</td>
</tr>
<tr>
<td></td>
<td><code>foreign-buffer</code> is a FLI pointer to a buffer, which must be of size at least <code>buffer-length</code>.</td>
</tr>
<tr>
<td>Notes</td>
<td><code>ora-lob-write-foreign-buffer</code> is available only when the &quot;oracle&quot; module is loaded. See the section “Oracle LOB interface” on page 380 for more information.</td>
</tr>
<tr>
<td>See also</td>
<td><code>ora-lob-get-buffer</code></td>
</tr>
<tr>
<td></td>
<td><code>ora-lob-write-buffer</code></td>
</tr>
</tbody>
</table>

**p-oci-env**

*FLI type descriptor*

**Summary** A foreign type representing objects in the Oracle interface.
The SQL Package

This chapter applies to the Enterprise Edition only

Package sql

Description See “Interactions with foreign calls” on page 384 for details.

**p oci-file**

*FLI type descriptor*

Summary A foreign type representing objects in the Oracle interface.

Package sql

Description See “Interactions with foreign calls” on page 384 for details.

**p oci-lob-locator**

*FLI type descriptor*

Summary A foreign type representing objects in the Oracle interface.

Package sql

Description See “Interactions with foreign calls” on page 384 for details.

**p oci-lob-or-file**

*FLI type descriptor*

Summary A foreign type representing objects in the Oracle interface.

Package sql

Description See “Interactions with foreign calls” on page 384 for details.

**p oci-svc-ctx**

*FLI type descriptor*

Summary A foreign type representing objects in the Oracle interface.
This chapter applies to the Enterprise Edition only

Package sql

Description See “Interactions with foreign calls” on page 384 for details.

prepare-statement

Function

Summary Returns a prepared-statement object for a sql-exp in a database.

Package sql

Signature

\[\text{prepare-statement } \text{sql-exp} \ & \text{key} \ \text{database} \ \text{variable-types} \ \text{count} \ \text{flatp} \ \text{result-types} \Rightarrow \text{prepared-statement}\]

Arguments

- sql-exp A SQL expression.
- database A database.
- variable-types A list.
- count A non-negative integer or nil.
- flatp A boolean.
- result-types A list of symbols.

Values

prepared-statement

A prepared-statement.

Description The function prepare-statement returns a prepared-statement object for the SQL statement sql-exp in the database database. sql-exp can contain bind-variables in the form :n where n is a positive integer.

If database is supplied, then the prepared-statement is associated with the database. Otherwise set-prepared-statement-variables will do the association even if it is called without a database.
If `variable-types` is supplied, then it should be a list containing a keyword element for each bind-variable in `sql-exp`. It has an effect in two cases:

- `:string` forces the variable to be passed to the database as a string. That may be useful if you have numeric values in Lisp which are stored as strings in the database.
- `:date` cause an integer to be interpreted as a universal-time and be converted properly to an Oracle date. This is not supported on SQLite databases, which do not support date fields.

If `variable-types` is not supplied, then the types will be chosen dynamically from the values passed to `set-prepared-statement-variables`.

If `count` is supplied, then it should equal the maximum number of bind-variables in the `sql-exp`. If `count` is not supplied, then it is calculated from `sql-exp`.

`flatp` and `result-types` are interpreted the same as in `select`.

The result of `prepare-statement` is a `prepared-statement`. This can be used by calling `set-prepared-statement-variables` to actually bind the variables, and then use one of the querying or executing interfaces that take a SQL expression argument: `execute-command`, `query`, `do-query`, `simple-do-query`, `map-query` and the `loop for...being each record` construct.

A `prepared-statement` that is associated with a database should be destroyed (by `destroy-prepared-statement`) before the database is closed, otherwise it may leak memory.

**Notes**

`sql-exp` can be any valid SQL expression, not only a query.

**Examples**

Create a `prepared-statement` for a SQL expression:

```lisp
(setq ps
  (sql:prepare-statement
   "insert into TABLETWO values(:1, :2")
  ))
```
Then insert records into TABLETWO (which has two columns) by repeatedly doing:

\[
\begin{align*}
&(\text{sql:set-prepared-statement-variables} \\
&\quad ps \\
&\quad (\text{list value1 value2})) \\
&(\text{sql:execute-command} ps)
\end{align*}
\]

See also

query
do-query
simple-do-query
map-query
select
set-prepared-statement-variables
destroy-prepared-statement

**prepared-statement**

*System Class*

**Summary**

A class of objects prepared SQL statements.

**Package**

sql

**Signature**

prepared-statement

**Description**

Each instance of the system class `prepared-statement` represents a SQL statement prepared by `prepare-statement`.

See also

prepare-statement

**print-query**

*Function*

**Summary**

Prints a tabulated version of records resulting from a query.

**Package**

sql
# The SQL Package

This chapter applies to the Enterprise Edition only

## Signature

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>print-query query-exp &amp;key titles formats sizes stream database =&gt;</code></td>
<td>Takes a symbolic SQL query expression and formatting information and prints onto a table containing the results of the query.</td>
</tr>
</tbody>
</table>

## Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>query-exp</code></td>
<td>A SQL query expression.</td>
</tr>
<tr>
<td><code>titles</code></td>
<td>A list of strings.</td>
</tr>
<tr>
<td><code>formats</code></td>
<td>A list of strings.</td>
</tr>
<tr>
<td><code>sizes</code></td>
<td>A list.</td>
</tr>
<tr>
<td><code>stream</code></td>
<td>An output stream.</td>
</tr>
<tr>
<td><code>database</code></td>
<td>A database.</td>
</tr>
</tbody>
</table>

## Values

None.

## Description

The function `print-query` takes a symbolic SQL query expression and formatting information and prints onto a table containing the results of the query.

A list of strings to use as column headings is given by `titles`, which has a default value of `nil`.

The `formats` argument is a list of format strings used to print each attribute, and has a default value of `t`, which means that `-A` or `-VA` are used if `sizes` are provided or computed.

The field sizes are given by `sizes`. It has a default value of `t`, which specifies that minimum sizes are computed.

The output stream is given by `stream`, which has a default value of `t`. This specifies that `*standard-output*` is used.

## Examples

The following call prints out two even columns of names and salaries:

```lisp
(print-query [select [surname] [income] :from [person]]
             :titles '("NAME" "SALARY")')
```

## See also

- `map-query`
- `print-query`
- `select`
<table>
<thead>
<tr>
<th>Summary</th>
<th>Queries a database and returns a list of values.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>sql</td>
</tr>
<tr>
<td>Signature</td>
<td><code>query sql-exp &amp;key database result-types flatp =&gt; result-list, field-names</code></td>
</tr>
<tr>
<td>Arguments</td>
<td>sql-exp</td>
</tr>
<tr>
<td></td>
<td>database</td>
</tr>
<tr>
<td></td>
<td>result-types</td>
</tr>
<tr>
<td></td>
<td>flatp</td>
</tr>
<tr>
<td>Values</td>
<td>result-list</td>
</tr>
<tr>
<td></td>
<td>field-names</td>
</tr>
<tr>
<td>Description</td>
<td>The function <code>query</code> is the basic SQL query function. It queries the database specified by <code>database</code> with a SQL query statement given by <code>sql-exp</code>. The argument <code>database</code> defaults to <code>*default-database*</code>. <code>result-types</code> is a list of symbols such as <code>:string</code> and <code>:integer</code>, one for each field in the query, which are used to specify the types to return. It is ignored if <code>sql-exp</code> is a prepared-statement. <code>flatp</code> is used as in <code>select</code> and is ignored if <code>sql-exp</code> is a prepared-statement. <code>result-list</code> is a list of values as per <code>select</code>, and <code>field-names</code> is a list of field names selected in <code>sql-exp</code>.</td>
</tr>
<tr>
<td>Example</td>
<td>The following two queries, on a table whose second column contains dates that we want to return as strings, are equivalent:</td>
</tr>
</tbody>
</table>
(sql:query "select * from some_table"
  :result-types '(nil :string))

(sql:query [select [*]
  :from [some_table]
  :result-types '(nil :string)])

See also

do-query
execute-command
lob-stream
loop
map-query
prepare-statement
select
simple-do-query

reconnect

Function

Summary

Reconnects a database to its underlying RDBMS.

Package

sql

Signature

reconnect &key database error force => success

Arguments

database The database to be reconnected.

error A boolean.

force A boolean.

Values

success A boolean.

Description

The function reconnect reconnects database to its underlying RDBMS. If successful, success is t and the variable *default-database* is set to the newly reconnected database.

The default value for database is *default-database*. If database is a database object, then it is used directly. Otherwise, the list of connected databases is searched to find
one with database as its connection specifications (see \texttt{connect}). If no such database is found, then if \texttt{error} and \texttt{database} are both non-nil an error is signaled, otherwise \texttt{reconnect} returns \texttt{nil}.

\textit{force} controls whether an error should be signaled if the existing database connection cannot be closed. When non-nil (this is the default value) the connection is closed without error checking. When \textit{force} is \texttt{nil}, an error is signaled if the database connection has been lost.

Notes \textit{force} non-nil might result in a memory leak if the database driver fails to release its memory (some drivers do not allow the connection to be closed if the underlying RDBMS is not responding).

See also \texttt{connect} \texttt{connected-databases} *\texttt{default-database}*

\texttt{restore-sql-reader-syntax-state} \textit{Function}

Summary Sets the enable/disable square bracket syntax state to reflect the last call to either \texttt{disable-sql-reader-syntax} or \texttt{enable-sql-reader-syntax}.

Package \texttt{sql}

Signature \texttt{restore-sql-reader-syntax-state}

Arguments None.

Values None.

Description The function \texttt{restore-sql-reader-syntax-state} sets the enable/disable state of the square bracket syntax to reflect the last call to either \texttt{enable-sql-reader-syntax} or \texttt{dis-}

1633
able-sql-reader-syntax. The default state of the square bracket syntax is disabled.

See also disable-sql-reader-syntax
enable-sql-reader-syntax
locally-disable-sql-reader-syntax
locally-enable-sql-reader-syntax

rollback  Function

Summary Rolls back changes made to a database since the last commit.

Package sql

Signature rollback &key database => nil

Arguments database A database.

Values nil

Description The function rollback rolls back changes made in database since the last commit, that is, changes made since the last commit are not recorded. The argument database defaults to *default-database*.

See also commit
with-transaction

select  Function

Summary Selects data from a database given a number of specified constraints.

Package sql
This chapter applies to the Enterprise Edition only

Signature

\[
\text{select} \ \&\text{rest} \ \text{selections} \ \&\text{key} \ \text{all} \ \text{set-operation} \ \text{distinct} \ \text{from} \\
\text{result-types} \ \text{flatp} \ \text{where} \ \text{group-by} \ \text{having} \ \text{database} \ \text{order-by} \ \text{refresh} \\
\text{for-update} \ =\Rightarrow \ \text{result-list}
\]

Arguments

- **selections**: A set of database identifiers or strings.
- **all**: A boolean.
- **set-operation**: A SQL operation.
- **distinct**: A boolean.
- **from**: A SQL table.
- **result-types**: A list of symbols.
- **flatp**: A boolean.
- **where**: A SQL condition.
- **group-by**: A SQL condition.
- **having**: A SQL condition.
- **database**: A database.
- **order-by**: A SQL condition.
- **refresh**: A boolean.
- **for-update**: \text{t, :nowait, a string or a list.}

Values

- **result-list**: A list of selections.

Description

The function \text{select} selects data from \text{database}, which has a default value of *default-database*, given the constraints specified by the rest of the arguments. It returns a list of objects as specified by \text{selections}. By default, the objects will each be represented as lists of attribute values.

The argument \text{selections} consists either of database identifiers, type-modified database identifiers or literal strings.

A type-modified database identifier is an expression such as \text{[foo :string]} which means that the values in column \text{foo} are returned as Lisp strings. This syntax can be used to force
values in time/date fields to be returned as strings (see below for an example). It can also be used to affect the value returned from MySQL, using the keywords mentioned in the section “Using MySQL” on page 373. It can also be used to return lob-stream objects for queries on Oracle LOB columns, using an expression like [foo :input-stream] or [foo :output-stream]

result-types is used when selections is * or [•]. It should be a list of symbols such as :string and :integer, one for each field in the table being selected in order to specify the types to return. Note that, for specific selections, the result type can be specified by using a type-modified identifier as described above. However, you cannot use result-types to modify the type returned from a time/date field.

flatp, which has a default value of nil, specifies if full bracketed results should be returned for each matched entry. If flatp is nil, the results are returned as a list of lists. If flatp is t, the results are returned as elements of a list, only if there is only one result per row. See the examples section for an example of the use of flatp.

The arguments all, set-operation, distinct, from, where, group-by, having and order-by have the same function as the equivalent SQL expression.

for-update is used to specify the FOR UPDATE clause in a select statement which is used by Oracle to lock the selected records. If for-update is t then a plain "FOR UPDATE" clause is generated. This locks all retrieved records, waiting for the locks to become available. If for-update is :nowait then a "FOR UPDATE NOWAIT" clause is generated. This locks all the retrieved records, or otherwise returns with error ora-00054 which causes Lisp to signal a sql-temporary-error. If for-update is a string then it should specify a column to be locked and a clause "FOR UPDATE OF for-update" is generated. If for-update is a list then the elements of the list should be strings each specifying a column to be locked,
except that the last element of the list may be :nowait. A clause locking multiple columns is generated, waiting for the locks according to whether :nowait was supplied. For an example see the section “Locking” on page 382.

The function select is common across both the functional and object-oriented SQL interfaces. If selections refers to View Classes then the select operation becomes object-oriented. This means that select returns a list of View Class instances, and slot-value becomes a valid SQL operator for use within the where clause.

In the View Class case, a second equivalent select call will return the same View Class instance objects. If refresh is true, then existing instances are updated if necessary, and in this case you might need to extend the hook instance-refreshed. Any join slots defined using retrieval:deferred will be recomputed the next time they are accessed. The default value of refresh is nil.

SQL expressions used in the select function are specified using the square bracket syntax, once this syntax has been enabled using enable-sql-reader-syntax.

SQL expressions used in the select function are commonly specified using the “Symbolic SQL syntax” on page 358. Note that you need to enable it by using enable-sql-reader-syntax or locally-enable-sql-reader-syntax so they can be read correctly. An expression can also be made dynamically by using sql-expression.

Examples

The following is a potential query and result:

```
(select [person_id] [surname] :from [person])
=> ((111 "Brown") (112 "Jones") (113 "Smith"))
```

In the next example, the flatp argument is set to t, and the result is a simple list of surname values:

```
(select [surname] :from [person] :flatp t)
```
=> (*"Brown" "Jones" "Smith")

In this example data in the attribute `largenum`, which is of a vendor-specific large numeric type, is returned to Lisp as strings:

```lisp
(sql:select [largenum :string] :from [my-table])
```

In this example the second column of `some_table` is a date that we want to return as a string:

```lisp
(sql:select [*]
    :from [some_table]
    :result-types '(nil :string))
```

In this example we see that a time/date field value is returned as an integer. We then use Common Lisp to decode that universal time, and finally query the database again, forcing the return value to be a string formatted by the database:

```lisp
CL-USER 219 > (sql:select [MyDate]
    :from [MyTable]
    :flatp t)
(3313785600)
(*MYDATE*)

CL-USER 220 > (decode-universal-time (car *))
0
0
0
4
1
2005
1
NIL
0

CL-USER 221 > (sql:select [MyDate :string]
    :from [MyTable]
    :flatp t)
(*"2005-01-04 00:00:00")
(*MYDATE*)
Finally this code gets the first 1 KB of data from the first LOB returned by a query on an Oracle table containing a column of type LOB:

\[
\begin{aligned}
&\text{(let* ((array (make-array 1024 :element-type '(unsigned-byte 8))))}
&\text{(lobs (sql:select [my-lob-column :input-stream]
&\text{ :from [mytable] :flatp t}))}
&\text{(read-sequence array (car lobs)))}
\end{aligned}
\]

See also
instance-refreshed
lob-stream
prepare-statement
print-query
sql-expression
query
“Symbolic SQL syntax” on page 358
“Querying” on page 346

### set-prepared-statement-variables

**Function**

**Summary**
Sets the values of the bind variables in a prepared-statement.

**Package**
sql

**Signature**

\[
\text{set-prepared-statement-variables prepared-statement &key}
\begin{align*}
database & \quad \text{values} \\
& \Rightarrow \text{prepared-statement}
\end{align*}
\]

**Arguments**

- **prepared-statement**
  A prepared-statement.
- **database**
  A database or nil.
- **values**
  A list.

**Description**
The function `set-prepared-statement-variables` sets the values of the bind variables in the prepared-statement pre-
pared-statement to the objects in the list given by values. The length of values must equal the number of bind-variables in prepared-statement (that is, the supplied or computed count in prepare-statement). If database is supplied, then prepared-statement is (re)associated with that database.

If database is not supplied and the statement is not associated with a database yet, set-prepared-statement-variables associates it with the default database *default-database*. If the statement was already associated and database is nil, the association does not change.

set-prepared-statement-variables returns the prepared-statement.

See also prepared-statement
prepare-statement
destroy-prepared-statement

**simple-do-query**

*Macro*

**Summary**
Repeatedly binds a variable to the results of a query, optionally binds another variable to the column names, and executes a body of code within the scope of these bindings.

**Package**
sql

**Signature**
simple-do-query (values-list query &key names-list database not-inside-transaction get-all) &body body =>

**Arguments**
values-list A variable.
query A database query or a prepared-statement containing a query.
names-list A variable, or nil.
database A database.
not-inside-transaction
This chapter applies to the Enterprise Edition only

A generalized boolean.

get-all A generalized boolean.

body A Lisp code body.

Values None.

Description The macro simple-do-query repeatedly executes body within a binding of values-list to the attributes of each record resulting from query.

If a variable names-list is supplied, then it is bound to a list of the column names for the query during the execution of body. The default value of names-list is nil.

simple-do-query returns no values.

The default value of database is *default-database*.

not-inside-transaction and get-all may be useful when fetching many records through a connection with database-type :mysql. Both of these arguments have default value nil. See the section “Special considerations for iteration functions and macros” on page 375 for details.

Example

```
(sql:simple-do-query
 (person-details [select [Surname][ID] :from [person]] :names-list xx)
 (format t "-&A: -A, -A: -A-%" (first xx) (first person-details) (second xx) (second person-details)))
=>
SURNAME: Brown, ID: 2
SURNAME: Jones, ID: 3
SURNAME: Smith, ID: 4
```

See also
do-query
loop
map-query
prepare-statement
The SQL Package

This chapter applies to the Enterprise Edition only.

query

select

sql

Function

Summary
Generates SQL from a set of expressions.

Package
sql

Signature
sql &rest args => sql-expression

Arguments
args
A set of expressions.

Values
sql-expression
A SQL expression.

Description
The function sql generates SQL from a set of expressions given by args. Each argument to sql is translated into SQL and then the args are concatenated with a single space between each pair. The rules for translation into SQL, based on the type of each individual argument x, are as follows:

string => (format nil "'~A'" x)
nil => "NULL"
symbol => (symbol-name x)
number => (princ-to-string x)
list => (format nil "(~{~A~^,~})" (mapcar #'sql x))
vector => (format nil "~{~A~^,~}" (map 'list #'sql x))

sql-expression => x

Any other symbol => error

See also
sql-expression
sql-operation
sql-operator
This chapter applies to the Enterprise Edition only

<table>
<thead>
<tr>
<th>Condition</th>
<th>Package</th>
<th>Superclasses</th>
<th>Subclasses</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sql-connection-error</strong></td>
<td>sql</td>
<td>sql-database-error</td>
<td>sql-fatal-error</td>
<td>The condition class <em>sql-connection-error</em> is used to signal an error with</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sql-timeout-error</td>
<td>the connection to the database.</td>
</tr>
<tr>
<td><strong>sql-database-data-error</strong></td>
<td>sql</td>
<td>sql-database-error</td>
<td></td>
<td>The condition class <em>sql-database-data-error</em> is used to signal an error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>with the data given. This means either a syntax error or things like</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>accessing a non-existent table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>It signifies an error that must be fixed for the code to work.</td>
</tr>
<tr>
<td><strong>sql-database-error</strong></td>
<td>sql</td>
<td>simple-error</td>
<td>sql-connection-error</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sql-database-data-error</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sql-temporary-error</td>
<td></td>
</tr>
</tbody>
</table>
Accessors  

- `sql-error-error-id`  
- `sql-error-secondary-error-id`  
- `sql-error-database-message`

Description  
The condition class `sql-database-error` is used to signal errors in the database interface that Common SQL uses.

- `sql-error-error-id` returns the primary error identifier. On ODBC the value is a string. On Oracle it is some number, the "v2 return code" in the Cursor Data Area.

- `sql-error-secondary-error-id` returns the secondary error identifier. On ODBC this is the error code from the underlying database. On Oracle that is the "v4 return code" (also known as "return code") in the Cursor Data Area, which is the useful code.

- `sql-error-database-message` is a string (maybe `nil`) that came back from the foreign code.

Notes  
ODBC drivers for Oracle return the "v4 return code" as the underlying database code. Therefore in the event of an error on connection to an Oracle database, `sql-error-secondary-error-id` always returns the "v4 return code" whether the connection is through ODBC.

See also  
`sql-user-error`

---

**sql-enlarge-static**  

*Variable*

Package  
`sql`

Initial value  
`100000`

Description  
The amount to enlarge static memory by before loading database code. This is an optimization of static memory fragmentation, useful for some databases. It is ignored when loading Oracle.
This chapter applies to the Enterprise Edition only

Notes  *sql-enlarge-static* is applicable in LispWorks for SPARC Solaris.

### sql-expression

**Function**

**Summary**
Generate a SQL expression from the supplied keywords.

**Package**
sql

**Signature**
```
sql-expression &key string table attribute owner alias type n-qualified parameter-index => sql-expression
```

**Arguments**
- `string` A string.
- `table` String, symbol or sql-expression-object.
- `attribute` String, symbol or sql-expression-object.
- `owner` String, symbol or sql-expression-object.
- `alias` A string.
- `type` A keyword.
- `n-qualified` A string.
- `parameter-index` Integer.

**Values**
- `sql-expression` A sql-expression-object.

**Description**
The function sql-expression generates a SQL expression from the supplied keywords. The result sql-expression is of type sql-expression-object, which specifies some SQL that can be used inside other calls to sql-expression, a "[...]" syntax expression or as an argument to the Common SQL functions. sql-expression matches what the read-time "[...]" syntax generates for identifiers (see “Enclosing database identifiers” on page 359), but can be used at run time.

If string is non-nil, it must be the only keyword and specifies the entire SQL, directly without any further processing.
If `attribute` is supplied, it specifies an attribute (column) name, and the resulting SQL is an attribute named by `attribute`. It can optionally be qualified by `table` and `owner` (the schema that owns the table), so the attribute name becomes `table.attribute` if `owner` is `nil` or `owner.table.attribute` if `owner` is non-nil. If `table` is `nil` the attribute name is not qualified.

If `table` is supplied, it specifies a table name. If `attribute` is supplied too, it qualifies the attribute as in the previous paragraph, otherwise the resulting SQL is a table name. It can be optionally qualified by `owner`, which specifies `owner.table` as the table name.

`owner` can be non-nil only when `table` is supplied, and qualifies `table` as described above. It specifies the schema to which the table belongs.

If `alias` is non-nil it specifies an alias, which added to the SQL after the attribute name if `attribute` is supplied or after the table name otherwise.

`type` can be non-nil only if `attribute` is supplied, and specifies the expected type of the attribute. This does not affect the SQL, but tells the querying interface (`select`, `query`) what type the value should be. It is useful when the attribute is used in the selection list of a query (see “Specifying the type of retrieved values.” on page 362).

If `parameter-index` is non-nil, it must be the only keyword, and `sql-expression` generates a bind-variable in the SQL. This can be used in an expression that is passed to `prepare-statement` and then later bound by `set-prepared-statement-variables`.

If `n-qualified` is non-nil, it must be the only keyword, and the value is used as-is as an N syntax string. This is required for passing non-ASCII string to Microsoft SQL Server (via ODBC), but does not work on SQLite or Microsoft Access.
This chapter applies to the Enterprise Edition only

```
(sql-expression :n-qualified "aa")
=> #<SQL: "N'aa'">

(sql-expression :string "aa")
=> #<SQL: "aa">
```

See “SQL string literals” on page 366 for discussion of N syntax strings.

**Examples**

Define a function that queries for the value of a supplied attribute in a supplied table using a supplied type:

```
(defun query-attribute (table-name attribute-name type)
  (let* ((table-arg
           (sql-expression
             :table table-name))
         (selection-arg
           (sql-expression
             :attribute attribute-name
             :type type)))
    (select selection-arg :from table-arg)))
```

See also:

- `sql`
- `sql-operation`
- `sql-operator`
- `sql-expression-object`
- “Symbolic SQL syntax” on page 358

**sql-expression-object**

*System Class*

**Summary**

A class of objects representing some SQL.

**Package**

`sql`

**Superclasses**

`t`

**Description**

Each instance of the system class `sql-expression-object` represents a SQL expression. They can be used inside another
instance of sql-expression-object, in a " [...]" syntax expression or as arguments to the Common SQL functions. They are created by the " [...]" syntax (see “Enclosing database identifiers” on page 359) or dynamically by sql-expression or string-prefix-with-n-if-needed.

See also sql-expression
“Enclosing database identifiers” on page 359

sql-fatal-error

Condition

Package sql

Superclasses sql-connection-error

Description The condition class sql-fatal-error is used to signal errors that mean the connection can no longer be used.

sqlite-blob

System Class

Summary A class of objects that provide read/write access to a BLOB or TEXT field in a SQLite database.

Package sql

Superclasses t

Description An instance of the system class sqlite-blob provides direct read/write access to a BLOB or TEXT field in a SQLite database. See replace-from-sqlite-blob for details.

See also replace-from-sqlite-blob
sqlite-raw-blob
sqlite-last-insert-rowid  

Function

Summary
Return the ROWID of the last inserted row in a SQLite database.

Package
sql

Signature
sqlite-last-insert-rowid &key database => rowid

Arguments
database A SQLite database.

Values
rowid An integer.

Description
The function `sqlite-last-insert-rowid` returns the ROWID of the last row that was inserted in `database`, or 0 if none.

Notes
`sqlite-last-insert-rowid` is not thread-safe, and you will need to ensure that no other thread inserts rows into `database` in parallel to the insertion of the row and calling `sqlite-last-insert-rowid`.

The result `rowid` is useful when you want to later access a BLOB in the row using `sqlite-open-blob`.

ROWIDs in SQLite are described in the SQLite documentation: "ROWIDs and the INTEGER PRIMARY KEY" in "CREATE TABLE" https://www.sqlite.org/lang_createtable.html#rowid.

Because `sqlite-last-insert-rowid` is called on the database connection, any row insertion into `database` affects it, even if it is not in the same table or even not the same file (if another file is attached to the connection using the "ATTACH DATABASE" statement). Therefore, there must not be another insertion into `database` in parallel to the sequence of insertion and calling `sqlite-last-insert-rowid`.

These restrictions mirror the underlying limitation of the C function `sqlite3_last_insert_rowid`.  

1649
The SQL Package

This chapter applies to the Enterprise Edition only

See also  
sqlite-open-blob

### Functions

#### Summary
Read/write access to a BLOB or TEXT field in a SQLite database.

#### Package
sql

#### Signature

```plaintext
sqlite-open-blob  table-name  column-name  rowid  &key  database  owner  read-only  =>  sqlite-blob
sqlite-close-blob  sqlite-blob  =>  boolean
sqlite-blob-p  object  =>  boolean
sqlite-blob-length  sqlite-blob  =>  length
replace-from-sqlite-blob  binary-array  sqlite-blob  &key  array-start  array-end  blob-start  blob-end  =>  binary-array
replace-into-sqlite-blob  sqlite-blob  binary-array  &key  blob-start  blob-end  array-start  array-end  =>  sqlite-blob
sqlite-reopen-blob  sqlite-blob  rowid
```

#### Arguments

- **table-name, column-name**
  - Strings.
- **rowid**
  - An integer.
- **database**
  - A SQLite database.
- **owner**
  - A string.
This chapter applies to the Enterprise Edition only

read-only A generalized boolean.
sqlite-blob An object of type sqlite-blob.
object Any object.
binary-array An array with integer or float element type, or a base-string, or a bmp-string.

array-start, array-end
Bounding index designators of binary-array.

blob-start, blob-end
Bounding index designators of sqlite-blob.

Values

sqlite-blob An object of type sqlite-blob.
boolean A boolean.
length An integer.
binary-array An array with integer or float element type, or a base-string, or a bmp-string.

Description

Instances of the system class sqlite-blob allow reading and writing from/to BLOB or TEXT fields in a SQLite database. It corresponds to the C structure sqlite3_blob (see "A Handle To An Open BLOB" in the SQLite documentation, https://www.sqlite.org/c3ref/blob.html).

The function sqlite-open-blob creates an object of type sqlite-blob, which can be used to access the data in a specific column and row of a SQLite database table, as specified by database, owner, table-name, column-name and rowid. owner specifies the schema-name (which defaults to "main"), and thus allows access to attached databases. table-name and column-name specify the table and column. rowid specifies the row where the value is. For documentation about rowid, see "ROWIDs and the INTEGER PRIMARY KEY" in "CREATE TABLE" in the SQLite documentation (https://www.sqlite.org/lang_createtable.html#rowid), and
also the notes below. read-only (which defaults to nil) specifies whether the result sqlite-blob is read-only or not.

The function sqlite-blob-p returns true if object is of type sqlite-blob and false otherwise.

The function sqlite-blob-length returns the length of sqlite-blob in bytes. Note that there is no way to change the length.

The functions replace-from-sqlite-blob and replace-into-sqlite-blob are used to copy from/to sqlite-blob, similar to replace or fli:replace-foreign-array. binary-array must be a binary array, which means an array of element type base-char, bmp-char, single-float, double-float, (unsigned-byte bit-size) or (signed-byte bit-size), where bit-size is one of 8, 16, 32 or (64-bit LispWorks only) 64. Note that simple-string is not regarded as a binary array, but bmp-string and base-string are. The length of sqlite-blob in elements is the length in bytes, as returned by the function sqlite-blob-length, truncated by the number bytes per element in binary-array. The values of array-start, array-end, blob-start and blob-end are all in elements (rather than bytes).

The function replace-from-sqlite-blob replaces the elements of binary-array between array-start and array-end by the elements of sqlite-blob between blob-start and blob-end. The function replace-into-sqlite-blob replaces in the other direction.

blob-start and array-start default to 0, array-end defaults to nil, meaning the length of binary-array, and blob-end defaults to nil, meaning the length of sqlite-blob in elements. When supplied, array-start must be a non-negative integer and not bigger than the length of binary-array, array-end must be not smaller than array-start and not bigger than the length of binary-array, blob-start must be a non-negative integer and not bigger than the length of sqlite-blob in elements, and blob-end must be not smaller than blob-start and not bigger than the length of sqlite-blob in elements. The number of elements cop-
This chapter applies to the Enterprise Edition only

ied is the smaller of the difference between blob-start and blob-end, and the difference between array-start and array-end.

replace-from-sqlite-blob and replace-into-sqlite-blob return their first argument.

The function sqlite-close-blob closes sqlite-blob and returns t if it closed, or nil if sqlite-blob was already closed.

The function sqlite-reopen-blob changes sqlite-blob to refer to a field in another row. In effect it closes sqlite-blob and reopens it with a different rowid but otherwise the same arguments as the sqlite-open-blob call that opened it.

Notes

You can obtain a ROWID by using rowid in the selection list of a query. For example, the following query returns a list of ROWIDs for records that match somecondition in the table SomeTable (in *default-database*):

```sql
(sql:select [rowid] :from [SomeTable] :where somecondition :flatp t)
```

The ROWID may be also be the value of a primary key in the table, as described in the SQLite documentation: "ROWIDs and the INTEGER PRIMARY KEY" in "CREATE TABLE" https://www.sqlite.org/lang_createtable.html#rowid.

It is also possible to find the ROWID of the last inserted row by sqlite-last-insert-rowid.

If the row where the field that sqlite-blob is accessing is modified, further access to sqlite-blob by replace-into-sqlite-blob or replace-from-sqlite-blob signals an error (of type sql-user-error). That is because SQLite itself does not allow further access. As a result, using sqlite-blob is not thread-safe, and you need be sure that no other code is trying to modify the same row while sqlite-blob is open.

sqlite-open-blob may fail for various reasons. When this happens, LispWorks retrieves the error message using the C function sqlite3_errmsg, which is not thread-safe (an
apparent misdesign of SQLite). As a result, you will get a misleading error message in very rare occasions, if another thread executing on the same database got an error in parallel. However, the error number, is always correct and its values are documented in the SQLite documentation "Result Code Meanings" https://www.sqlite.org/res-code.html#error.

Leaving a sqlite-blob opened is not only a resource leak, but also leaves some locks in the database connection that prevents some operations in the future (dropping the table or disconnecting the database for example). Therefore, you should close a sqlite-blob as soon as possible. We recommend using with-sqlite-blob to open and close the sqlite-blob when possible.

See also
- with-sqlite-blob
- sqlite-blob
- “Using SQLite” on page 392

sqlite-raw-blob

System Class

Summary
A class of objects that allow efficient access to a SQLite BLOB.

Package
sql

Superclasses
t

Description
An instance of the system class sqlite-raw-blob allows you to efficiently access SQLite BLOB objects inside the dynamic extent of the Common SQL iterative querying interface.

See copy-from-sqlite-raw-blob for details.

See also
- copy-from-sqlite-raw-blob
- sqlite-blob
This chapter applies to the Enterprise Edition only

sqlite-raw-blob-p
sqlite-raw-blob-valid-p
sqlite-raw-blob-length
sqlite-raw-blob-ref
copy-from-sqlite-raw-blob
replace-from-sqlite-raw-blob

**Functions**

**Summary**
Efficient access to a SQLite BLOB field in a query.

**Package**
sql

**Signature**

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlite-raw-blob-p</td>
<td>object =&gt; boolean</td>
</tr>
<tr>
<td>sqlite-raw-blob-valid-p</td>
<td>sqlite-raw-blob =&gt; boolean</td>
</tr>
<tr>
<td>sqlite-raw-blob-length</td>
<td>sqlite-raw-blob =&gt; length</td>
</tr>
<tr>
<td>sqlite-raw-blob-ref</td>
<td>sqlite-raw-blob index &amp;optional foreign-element-type =&gt; value</td>
</tr>
<tr>
<td>copy-from-sqlite-raw-blob</td>
<td>&amp;key start end element-type =&gt; binary-array</td>
</tr>
<tr>
<td>replace-from-sqlite-raw-blob</td>
<td>binary-array sqlite-raw-blob &amp;key array-start array-end blob-start blob-end =&gt; binary-array</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Any Lisp object.</td>
</tr>
<tr>
<td>sqlite-raw-blob</td>
<td>An object of type sqlite-raw-blob.</td>
</tr>
<tr>
<td>index</td>
<td>An integer.</td>
</tr>
<tr>
<td>foreign-element-type</td>
<td>A foreign element type.</td>
</tr>
<tr>
<td>start, end</td>
<td>Bounding index designators of sqlite-raw-blob.</td>
</tr>
<tr>
<td>element-type</td>
<td>A Lisp element type for a binary array.</td>
</tr>
<tr>
<td>binary-array</td>
<td>An array with integer or float element type or a base-string or a bmp-string,</td>
</tr>
</tbody>
</table>
array-start, array-end

Bounding index designators of binary-array.

blob-start, blob-end

Bounding index designators of sqlite-raw-blob.

Values

- **boolean**: A boolean.
- **length**: An integer.
- **value**: A Lisp value of type matching foreign-element-type.
- **binary-array**: An array with integer or float element type or a base-string or a bmp-string.

Description

The sqlite-raw-blob interface allows a flexible and more efficient way to read from a SQLite BLOB object inside the dynamic extent of a Common SQL the iterative querying interface. The iterative querying interfaces include map-query, do-query, simple-do-query, loop with each record, but does not include select and query.

Note that sqlite-raw-blob corresponds to the result of the C function sqlite3_column_blob. You can read data from a BLOB using sqlite-raw-blob, but cannot modify it. For an interface that corresponds to the C structure sqlite3_blob, see sqlite-blob.

You receive a sqlite-raw-blob object as the value from the query for fields where you specify the type as :blob. This object is associated with a SQLite BLOB corresponding to the value of this field in the current row. For example, you can print the size (in bytes) of all the fields in the DataPointsColumn in SomeTable using this code:
This chapter applies to the Enterprise Edition only

```lisp
(do-query
  ((raw-data-points)
   [select [DataPointsColumn :blob]
    :from [Sometable]])
  (print (sqlite-raw-blob-length raw-data-points)))
```

The function `sqlite-raw-blob-p` returns true if `object` is of type `sqlite-raw-blob` and false otherwise.

The function `sqlite-raw-blob-length` returns the length in bytes of the BLOB associated with `sqlite-raw-blob`.

For the functions `sqlite-raw-blob-ref`, `copy-from-sqlite-raw-blob` and `replace-from-sqlite-raw-blob`, `sqlite-raw-blob` can be regarded as a handle to a foreign array of bytes, whose length in bytes as returned by `sqlite-raw-blob-length`. When the element type argument (foreign-element-type or element-type) requires more than one byte, then the length of `sqlite-raw-blob` in elements is the length in bytes truncated by the number of bytes per element.

The function `sqlite-raw-blob-ref` is analogous to `fli:dereference`. The element type is specified by foreign-element-type, which defaults to `(unsigned :byte)`. `index` must be a non-negative integer and smaller than the length of `sqlite-raw-blob` in elements. `sqlite-raw-blob-ref` returns value in the same way that `(fli:dereference pointer-to-the-blob-data :index index :type foreign-element-type)` would return it, if pointer-to-the-blob-data pointed to a foreign array with the same contents as `sqlite-raw-blob`.

The function `copy-from-sqlite-raw-blob` returns a Lisp array of element type `element-type` containing the elements of `sqlite-raw-blob` bounded by `start` and `end` (in `start` and `end` specify the start and the end indices in elements (rather than bytes) into `sqlite-raw-blob`. `start` defaults to 0, and `end` to `nil`, meaning the length of `sqlite-raw-blob` in elements. When supplied, `start` must be a non-negative integer and not bigger than the length of `sqlite-raw-blob` in elements, and `end` must be not smaller than `start` and not bigger than the length of `sqlite-raw-blob` in elements. `element-type` specifies the Lisp type of
binary-array. It is upgraded by upgraded-array-element-type, and the result must be one of:

- base-char
- bmp-char
- single-float
- double-float
- (unsigned-byte 8)
- (unsigned-byte 16)
- (unsigned-byte 32)
- (unsigned-byte 64) (64-bit LispWorks only)
- (signed-byte 8)
- (signed-byte 16)
- (signed-byte 32)
- (signed-byte 64) (64-bit LispWorks only)

The function replace-from-sqlite-raw-blob is analogous to fli:replace-foreign-array. It replaces the elements of binary-array, bounded by array-start and array-end, by the elements of sqlite-raw-blob, bounded by blob-start and blob-end (all in elements). binary-array must be a binary array, which means an array with one of the element types listed in the previous paragraph. array-start and blob-start default to 0, array-end defaults to nil, meaning the length of binary-array, and blob-end defaults to nil, meaning the length of sqlite-raw-blob in elements. When supplied, array-start must be a non-negative integer and not bigger than the length of binary-array, array-end must be not smaller than array-start and not bigger than the length of binary-array, blob-start must be a non-negative integer and not bigger than the length of sqlite-raw-blob in elements, and blob-end must be not smaller than blob-start and not bigger than the length of sqlite-raw-blob in elements. The number of elements copied is the smaller of
the difference between array-start and array-end, and the difference between blob-start and blob-end. replace-from-sqlite-raw-blob returns binary-array.

A sqlite-raw-blob object is valid only inside the dynamic extent of the code that receives it from the iterative querying interface function or macro. Note that the sqlite-raw-blob is already invalid in the next iteration of the same operation. Trying to read data from an invalid sqlite-raw-blob using one of sqlite-raw-blob-ref, copy-from-sqlite-raw-blob or replace-from-sqlite-raw-blob signals an error (of type sql-user-error). sqlite-raw-blob-length still returns the correct value for an invalid sqlite-raw-blob. sqlite-raw-blob-valid-p can be used to check if a sqlite-raw-blob is valid, but should be rarely useful.

See also

map-query
do-query
simple-do-query
loop
sqlite-raw-blob
sqlite-blob

“Iteration” on page 351

*sql-libraries*

<table>
<thead>
<tr>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package</strong></td>
</tr>
<tr>
<td><strong>Initial value</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
</tbody>
</table>
value is a list, then it is assumed to be a complete list of full library names which are loaded verbatim.

Notes  *sql-libraries* is applicable only on Unix/Linux.

**sql-loading-verbose**

<table>
<thead>
<tr>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
</tr>
<tr>
<td>Initial value</td>
</tr>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Notes</td>
</tr>
</tbody>
</table>

**sql-operation**

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
</tr>
<tr>
<td>Package</td>
</tr>
<tr>
<td>Signature</td>
</tr>
</tbody>
</table>

```
sql-operation op &rest args => sql-result
sql-operation sql-function name &rest args => sql-result
sql-operation sql-operator inop1 left &rest rights => sql-result
sql-operation sql-boolean-operator inop2 left &rest rights => sql-result
```

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
</tr>
<tr>
<td>args</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>args</td>
</tr>
</tbody>
</table>
This chapter applies to the Enterprise Edition only

| inop1       | An infix operator with non-boolean result. |
| inop2       | An infix operator that returns a boolean.  |
| left        | Argument to be placed on the left of an infix operator. |
| rights      | Arguments to be placed on the right of an infix operator. |

Values  

| sql-result | A SQL expression. |

Description  
The function **sql-operation** takes an operator and its arguments, and returns a SQL expression.

\[(sql-operation \textit{op} \textit{args})\]

is shorthand for

\[(apply (sql-operator \textit{op}) \textit{args}).\]

The pseudo operator **sql-function** allows an arbitrary function **name** to be passed. In this case, **name** is put in the SQL expression using **princ**, and **args** are given as arguments.

The pseudo operators **sql-boolean-operator** and **sql-operator** generate SQL that calls an infix operator with **left** on the left and **rights** on the right separated by spaces. Use **sql-boolean-operator** for SQL infix operators that return a boolean and use **sql-operator** for any other SQL infix operator.

Notes  
The pseudo operator **sql-operator** should not be confused with the Common SQL function **sql-operator**.

Example  
The following code, uses **sql-operation** to produce a SQL expression.
(sql-operation 'select
  (sql-expression :table 'foo :attribute 'bar)
  (sql-expression :attribute 'baz)
 :from (list
    (sql-expression :table 'foo)
    (sql-expression :table 'quux))
 :where
  (sql-operation 'or
    (sql-operation '>
      (sql-expression :attribute 'baz)
      3)
    (sql-operation 'like
      (sql-expression :table 'foo :attribute 'bar)
      "SU\%"))

The following SQL expression is produced.

#<SQL-QUERY: "(SELECT FOO.BAR,BAZ FROM FOO,QUUX
  WHERE ((BAZ > 3) OR (FOO.BAR LIKE 'SU\%')))">

The following code illustrates use of the pseudo operator
sql-function:

(sql-operation 'sql-function "TO_DATE" "03/06/99"
  "mm/DD/RR")

The following SQL expression is produced.

#<SQL-VALUE-EXP "TO_DATE('03/06/99','mm/DD/RR')">

See also
sql
sql-expression
sql-operator

sql-operator

Function

Summary
Returns the symbol for a SQL operator.

Package
sql

Signature
sql-operator symbol => sql-symbol

Arguments
symbol
A symbol naming a SQL operator.
Values  

| sql-symbol | A symbol. |

Description  
The function `sql-operator` takes an operator as an argument and returns the Lisp symbol for the operator.

See also  
`sql`  
`sql-expression`  
`sql-operation`

---

**sql-recording-p**  

*Function*

Summary  
A predicate for determining if SQL commands or results traffic is being recorded.

Package  
`sql`

Signature  
`sql-recording-p &key type database => recording-p`

Arguments  
`type`  
One of :commands or :results.  
`database`  
A database.

Values  
`recording-p`  
A boolean.

Description  
The function `sql-recording-p` returns `t` if `type` is :commands and SQL commands traffic is being recorded, or if `type` is :results and SQL results traffic is being recorded. Otherwise it returns `nil`.  

The default value of `type` is :commands. The default value of `database` is the value of `*default-database*`.

See also  
`add-sql-stream`  
`delete-sql-stream`  
`list-sql-streams`  
`sql-stream`
Function

sql-stream

Summary
Returns the broadcast stream used for recording SQL commands or results traffic

Package
sql

Signature
sql-stream &key type database => stream

Arguments
- type: One of :commands or :results.
- database: A database.

Values
- stream: A broadcast stream.

Description
The function sql-stream returns the broadcast stream used for recording SQL commands or results traffic.

type can be either :commands or :results, and specifies whether to return the broadcast stream for commands or results traffic.

The default value of type is :commands. The default value of database is the value of *default-database*.

Note that SQL traffic can appear on *standard-output* as well as on stream. See add-sql-stream for details.

See also
- add-sql-stream
- delete-sql-stream
- list-sql-streams
- sql-recording-p
- start-sql-recording
- stop-sql-recording
This chapter applies to the Enterprise Edition only

**sql-temporary-error**

*Condition*

Package   sql

Superclasses   sql-database-error

Description   The condition class *sql-temporary-error* is used to signal an error that results from other users using the same database. This can be a table lock, but also running out of various resources. It means the code can work without change, once the other users stop using the database.

**sql-timeout-error**

*Condition*

Package   sql

Superclasses   sql-connection-error

Description   The condition class *sql-timeout-error* is used to signal an error due to the time out of some operation.

**sql-user-error**

*Condition*

Package   sql

Superclasses   simple-error

Description   The condition class *sql-user-error* is used to signal errors in Lisp code.

See also   sql-database-error
**standard-db-object**

**Class**

- **Package**: `sql`
- **Superclasses**: `standard-object`
- **Description**: The class `standard-db-object` implements View Classes.
- **See also**: `def-view-class`

**start-sql-recording**

**Function**

- **Summary**: Starts recording SQL commands or results traffic.
- **Package**: `sql`
- **Signature**: `start-sql-recording &key type database =>`
- **Arguments**:
  - `type`: A keyword.
  - `database`: A database.
- **Values**: None.
- **Description**: The function `start-sql-recording` starts recording SQL traffic, potentially to multiple streams. The traffic recorded can be the commands, the results, or both commands and results.

  By default the output appears only `*standard-output*`. You can modify the broadcast list of recording streams using `add-sql-stream` and `delete-sql-stream`.

  `type` is one of `:commands`, `:results` or `:both`. It determines whether SQL commands traffic, results traffic or both is recorded.
This chapter applies to the Enterprise Edition only

The default value of \texttt{type} is \texttt{:commands}. The default value for \texttt{database} is the value of *\texttt{default-database}*.

See also add-sql-stream
delete-sql-stream
list-sql-streams
sql-stream
sql-recording-p
stop-sql-recording

\textbf{status} \hspace{1cm} \textit{Function}

\textbf{Summary} Returns status information for the connected databases and initialized database types.

\textbf{Package} sql

\textbf{Signature} \texttt{status &optional full =>}

\textbf{Arguments} \texttt{full} \hspace{1cm} A boolean.

\textbf{Values} None.

\textbf{Description} The function \texttt{status} prints status information to the standard output, for the connected databases and initialized database types.

If \texttt{full} is \texttt{t}, detailed status information is printed. The default value of \texttt{full} is \texttt{nil}.

See also connect
connected-databases
database-name
disconnect
find-database
**stop-sql-recording**

**Function**

**Summary**
Stops recording SQL commands or results traffic.

**Package**
sql

**Signature**
\[
\text{stop-sql-recording} \ &\text{key type database} \Rightarrow
\]

**Arguments**
- **type**: A keyword.
- **database**: A database.

**Values**
None.

**Description**
The function \texttt{stop-sql-recording} stops recording SQL commands or results traffic.

\textit{type} is one of \texttt{:commands}, \texttt{:results} or \texttt{:both}. It determines whether the recording of SQL commands traffic, results traffic or both is stopped.

The default value of \textit{type} is \texttt{:commands}. The default value for \textit{database} is \texttt{*default-database*}.

**See also**
- add-sql-stream
- delete-sql-stream
- list-sql-streams
- sql-recording-p
- sql-stream
- start-sql-recording

**string-needs-n-prefix**

**Function**

**Summary**
Returns whether a string needs the N syntax.

**Package**
sql

**Signature**
\[
\text{string-needs-n-prefix} \ &\text{key database} \Rightarrow \text{needs-n-prefix-p}
\]
This chapter applies to the Enterprise Edition only

Arguments

- **string**: A string.
- **database**: A database.

Values

- **needs-n-prefix-p**: A boolean.

Description

The function **string-needs-n-prefix** returns true if **string** needs to be prefixed by N when passed to **database** (default *default-database*).

Notes

The function **string-prefix-with-n-if-needed** can be used to add the prefix if needed. The function **sql-expression** with **:n-qualified** can be used to unconditionally add the prefix.

At the time of writing, the prefix is required only when **database** is a connection to Microsoft SQL Server, and **string** contains characters which are not recognized by the code page of the server.

See also

- **string-prefix-with-n-if-needed**
- **sql-expression**
- “SQL string literals” on page 366

### **string-prefix-with-n-if-needed**

**Function**

**Summary**

Adds the N syntax to a string if needed.

**Package**

sql

**Signature**

```
string-prefix-with-n-if-needed string &key database => result
```

**Arguments**

- **string**: A string.
- **database**: A database.

**Values**

- **result**: A string or a **sql-expression-object**.
The SQL Package

Description
The function `string-prefix-with-n-if-needed` checks if `string`, when passed to `database`, needs to be prefixed by N. If the prefix is required, it returns a `sql-expression-object` with the string prefixed. Otherwise it returns `string`.

Notes
`string-prefix-with-n-if-needed` is equivalent to:

```
(if (string-needs-n-prefix string :database database)
    (sql-expression :n-qualified string)
    string)
```

See also
“SQL string literals” on page 366

string-needs-n-prefix

---

**table-exists-p**

*Function*

**Summary**
A predicate for the existence of a table.

**Package**
sql

**Signature**
`table-exists-p table &key database owner => result`

**Arguments**
- `table` A potential table name.
- `database` A database.
- `owner` `nil`, `:all` or a string.

**Values**
- `result` A boolean.

**Description**
The function `table-exists-p` determines whether there is a table named `table` in database `database`.

If `owner` is `nil`, only user-owned tables are considered. This is the default.

If `owner` is `:all`, all tables are considered.

If `owner` is a string, this denotes a username and only tables owned by `owner` are considered.
The default value of *database* is *default-database*.

See also list-tables

**update-instance-from-records**

*Generic Function*

**Summary**

Updates a View Class instance.

**Package**

sql

**Signature**

update-instance-from-records instance &key database => instance

**Arguments**

*instance*  An instance of a View Class.

*database*  A database.

**Values**

*instance*  The updated View Class instance.

**Description**

The generic function *update-instance-from-records* updates the values in the slots of the View Class instance *instance* using the data in the database *database*.

*database* defaults to the database that *instance* is associated with, or the value of *default-database*. If *instance* is associated with a database, then *database* must be that same database.

The argument *slot* is the CLOS slot name; the corresponding column names are derived from the View Class definition.

The update is not recursive on joins. Join slots (that is, slots with :db-kind :join) are updated, but the joined objects are not updated.

See also def-view-class

update-slot-from-record
**update-objects-joins**

**Summary**
Updates the remote join slots.

**Signature**

```
update-objects-joins objects &key slots force-p class-name max-len
```

**Arguments**
- `objects` A list of database objects.
- `slots` A list of slot names, or `t`.
- `force-p` A boolean.
- `class-name` The class of the objects, or `nil`.
- `max-len` A non-negative integer, or `nil`.

**Description**

The function `update-objects-joins` updates the remote join slots, that is those slots defined without `:retrieval :immediate`.

This is an optimization function which can improve the efficiency of an application by reducing the number of queries of the database. For each slot, it queries the database using the data from all the objects, and then assigns the appropriate value to each object.

`objects` is a list of database objects. If `class-name` is non-nil, then all the database objects are of this class. If `class-name` is `nil`, then all the database objects are of the class of the first database object in the list `objects`.

If `objects` is `nil`, then `update-objects-joins` does nothing.

`class-name` specifies a class containing all the database objects in the list `objects`. If `class-name` is `nil` (the default) then the class of the first database object is used.

`slots` provides a list of the names of slots to update. Each of these slots should be a remote join slot (as defined above).

`slots` can also be `t`, meaning update all the remote join slots. The default value of `slots` is `t`. 
This chapter applies to the Enterprise Edition only

force-p controls whether to force the update of all values in the objects. If force-p is nil, then slots which are already are not updated. The default value of force-p is t.

max-len, if non-nil, is a maximum number of objects from which to use data in a single query. If the length of the list objects is greater than max-len then update-objects-joins performs multiple queries using the data from no more than max-len objects in each query. This is useful if the DBMS may reject large queries, but it will increase the number of queries and hence reduce overall performance to some extent. The default value of max-len is the value of the variable *default-update-objects-max-len*.

See also
*default-update-objects-max-len*
def-view-class

**update-records**

*Function*

**Summary**
Changes the values of fields in a table.

**Package**
sql

**Signature**

update-records table &key attributes values av-pairs where database =>

**Arguments**

table A database table.

attributes A set of columns.

values A set of values.

av-pairs An association list alternative to attributes and values.

where A condition.

database A database.

**Values**

None.
The function `update-records` changes the values of existing fields in `table` with columns specified by `attributes` and `values` (or `av-pairs`) where the `where` condition is true.

**See also**
- `delete-instance-records`
- `delete-records`
- `insert-records`
- `update-records-from-instance`

### `update-records-from-instance`  
**Generic Function**

**Summary**
Updates a set of specified records in a database.

**Package**
`sql`

**Signature**
`update-records-from-instance` `instance` `&key` `database` `=>`

**Arguments**
- `instance` An instance of a View Class.
- `database` A database.

**Values**
None.

**Description**
The generic function `update-records-from-instance` updates the records in `database` represented by `instance`. If the instance is already associated with a database, that database is used, and `database` is ignored. If `instance` is not yet associated with a database, a record is created for `instance` in the appropriate table of `database` and the instance becomes associated with that database.

`update-records-from-instance` only updates the records from the base slots of `instance` - it does not look at the join slots.
This chapter applies to the Enterprise Edition only

See also
- def-view-class
- delete-instance-records
- update-records

**update-record-from-slot**

*Generic Function*

**Summary** Updates an individual data item from a slot.

**Package** sql

**Signature** update-record-from-slot instance slot &key database

**Arguments**
- *instance* An instance of a View Class.
- *slot* A slot.
- *database* A database.

**Values** None.

**Description** The generic function `update-record-from-slot` updates an individual data item in the column represented by `slot`. The `database` is only used if `instance` is not yet associated with any database, in which case a record is created in `database`. Only `slot` is initialized in this case; other columns in the underlying database receive default values. The argument `slot` is the CLOS slot name; the corresponding column names are derived from the View Class definition.

See also
- def-view-class
- update-records-from-instance

**update-slot-from-record**

*Generic Function*

**Summary** Updates a slot in a View Class instance.
The SQL Package

This chapter applies to the Enterprise Edition only

<table>
<thead>
<tr>
<th>Package</th>
<th>sql</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>update-slot-from-record instance slot =&gt; instance</td>
</tr>
<tr>
<td>Arguments</td>
<td>instance</td>
</tr>
<tr>
<td></td>
<td>slot</td>
</tr>
<tr>
<td>Values</td>
<td>instance</td>
</tr>
<tr>
<td>Description</td>
<td>The generic function update-slot-from-record updates the value in the slot slot of the View Class instance instance using the records in the database.</td>
</tr>
<tr>
<td></td>
<td>instance</td>
</tr>
<tr>
<td></td>
<td>The argument slot is the CLOS slot name; the corresponding column names are derived from the View Class definition.</td>
</tr>
<tr>
<td></td>
<td>The update is not recursive on joins. Join slots (that is, slots with :db-kind :join) are updated, but the joined objects are not updated.</td>
</tr>
<tr>
<td>See also</td>
<td>def-view-class</td>
</tr>
<tr>
<td></td>
<td>update-instance-from-records</td>
</tr>
</tbody>
</table>

**use-n-syntax-for-non-ascii-strings**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Control whether the symbolic SQL syntax uses the N syntax for non-ASCII SQL string literals.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>sql</td>
</tr>
<tr>
<td>Initial Value</td>
<td>nil</td>
</tr>
<tr>
<td>Description</td>
<td>The variable <em>use-n-syntax-for-non-ascii-strings</em> controls whether SQL string literals containing non-ASCII characters are put into SQL expressions with the N syntax.</td>
</tr>
</tbody>
</table>
This chapter applies to the Enterprise Edition only

When *use-n-syntax-for-non-ascii-strings* is nil (the default), all string literals are produced without the N syntax. When *use-n-syntax-for-non-ascii-strings* is non-nil, non-ASCII string literals are produced with the N syntax.

A non-ASCII string is any string that contains character codes out of the ASCII range 0 to 127.

For example:

```
(sql:sql-operation '=' [name] "hhh<Greek>")
=> #<SQL-RELATIONAL-EXP "(NAME = 'hhh<Greek>')">

(let ((sql:*use-n-syntax-for-non-ascii-strings* t))
  (sql:sql-operation '=' [name] "hh<Greek>"))
=> #<SQL-RELATIONAL-EXP "(NAME = N'hh<Greek>')">
```

For the symbolic SQL "[...]" syntax, the effect of *use-n-syntax-for-non-ascii-strings* occurs at macro expansion time. Therefore, if you use the symbolic SQL syntax and want to make of use *use-n-syntax-for-non-ascii-strings*, then you need to set it before compiling your code.

See “SQL string literals” on page 366 for details.

Notes

Microsoft SQL Server is currently the only SQL backend that requires the N syntax.

*use-n-syntax-for-non-ascii-strings* does not affect what sql-expression with :string does.

**with-sqlite-blob**

*Macro*

**Summary**

Execute code with an open handle to a BLOB in SQLite.

**Package**

sql
Signature

```
with-sqlite-blob (blob-var table-name column-name rowid &key
database owner read-only) &body body => results-of-body
```

Arguments

- `blob-var`: A symbol.
- `table-name, column-name`: Strings.
- `rowid`: An integer.
- `database`: A SQLite database.
- `owner`: A string.
- `read-only`: A generalized boolean.
- `body`: Lisp forms.

Values

- `results-of-body`: Multiple values of any Lisp type.

Description

The macro `with-sqlite-blob` opens a BLOB by calling `sqlite-open-blob` with `database`, `owner`, `table-name`, `column-name`, `rowid` and `read-only`, binds `blob-var` to the resulting `sqlite-blob`, and evaluates the forms in `body` (as an implicit `progn`) inside the binding and inside an `unwind-protect` that closes `blob-var` on exit.

The return values of `with-sqlite-blob`, `results-of-body`, are whatever `body` returns.

Notes

Because `with-sqlite-blob` guarantees to close the BLOB when it exits, you should use it in preference to calling `sqlite-open-blob` directly.

See also

`sqlite-open-blob`

### with-transaction

**Macro**

**Summary**

Performs a body of code within a transaction for a database.
This chapter applies to the Enterprise Edition only

Package  sql

Signature  with-transaction &key database &body body => results

Arguments  database  A database.
body  A set of Lisp expressions.

Values  results  The values returned by body.

Description  The macro with-transaction executes body within a transaction for database (which defaults to *default-database*). The transaction is committed if the body finishes successfully (without aborting or throwing), otherwise the database is rolled back.

with-transaction returns the value or multiple values returned from body.

Example  The following example shows how to use with-transaction to insert a new record, updates the department number of employees from 40 to 50, and removes employees whose salary is higher than 300,000. If an error occurs anywhere in the body and an abort or throw is executed, none of the updates are committed.

(with-transaction
  (insert-record :into [emp]
    :attributes '(x y z)
    :values '(a b c))
  (update-records [emp]
    :attributes [dept]
    :values 50
    :where [= [dept] 40])
  (delete-records :from [emp]
    :where [> [salary] 300000])))

See also  commit
rollback
This chapter applies to the Enterprise Edition only
This chapter describes the symbols available in the `stream` package that provide users with the functionality to define their own streams for use by the standard I/O functions.

This is discussed in detail in Chapter 24, “User Defined Streams”.

**buffered-stream**

**Summary**
A stream class giving access to stream buffers.

**Package**
stream

**Superclasses**
fundamental-stream

**Subclasses**
lob-stream
string-stream
socket-stream

**Initargs**
:direction One of :input, :output or :io. This argument is required.
:element-type One of base-char, bmp-char, simple-char or character.

Description

The class buffered-stream provides default methods for the majority of the functions in the User Defined Streams protocol. The default methods implement buffered I/O, requiring the user to define only the methods stream-read-buffer, stream-write-buffer and stream-element-type for each subclass of buffered-stream. You are at liberty to redefine other methods in subclasses as long as they obey the rules outlined here. For example it is usually desirable to implement methods on stream-listen, stream-check-eof-no-hang and close as well.

The initargs are handled by the method (method initialize-instance :after (buffered-stream)) as follows:

Input and/or output buffers are created based on the value direction. There is no default value, and you must supply a value.

element-type determines the stream-element-type of the stream. The default is base-char. For binary streams, use base-char.

All the methods in the User Defined Streams protocol are defined for buffered-stream as follows:

- The methods on stream-read-char, stream-read-line, stream-read-sequence, stream-unread-char, stream-read-char-no-hang, stream-clear-input handle input from the buffer. They each call stream-fill-buffer to fill the empty buffer as required.

- The methods on stream-write-char, stream-write-string, stream-write-sequence, stream-clear-output, stream-finish-output, stream-force-output and stream-line-column handle output to the buffer. They each call stream-flush-buffer to make the buffer empty as required.
• There are :around methods on stream-listen and close which handle the buffer.

• The methods on input-stream-p, output-stream-p return the appropriate values based on the value of the :direction initarg.

• The open-stream-p method returns true if close has not been called.

Example
See the extended example in
(exexample-edit-file "streams/buffered-stream")

See also
close
stream-flush-buffer
stream-fill-buffer
stream-listen
stream-read-buffer
stream-write-buffer
with-stream-input-buffer

fundamental-binary-input-stream

Class

Summary
A stream class for binary input.

Package
stream

Superclasses
fundamental-binary-stream
fundamental-input-stream

Subclasses
None.

Description
The class fundamental-binary-input-stream provides a class for generating customized binary input stream classes. A method for stream-read-byte should be provided when using this class.
See also  
* fundamental-binary-stream  
* fundamental-input-stream  
* stream-read-byte  

---

**fundamental-binary-output-stream**  
*Class*

**Summary**  
A stream class for binary output.

**Package**  
*stream*

**Superclasses**  
*fundamental-binary-stream*  
*fundamental-output-stream*

**Description**  
The class *fundamental-binary-output-stream* provides a class for generating customized binary output stream classes. A method for *stream-write-byte* should be provided.

See also  
* fundamental-binary-stream  
* fundamental-output-stream  
* stream-write-byte  

---

**fundamental-binary-stream**  
*Class*

**Summary**  
A class for binary streams.

**Package**  
*stream*

**Superclasses**  
*fundamental-stream*

**Subclasses**  
*fundamental-binary-input-stream*  
*fundamental-binary-output-stream*

**Description**  
The class *fundamental-binary-stream* is the superclass of the binary input and output stream classes. A method for *stream-element-type* should be provided for concrete subclasses of this class.
### fundamental-character-input-stream

**Class**

| Summary | A class that should be included in stream classes for character input. |
| Package | stream |
| Superclasses | fundamental-character-stream  
fundamental-input-stream |
| Subclasses | None. |
| Description | The class `fundamental-character-input-stream` provides default methods for generic functions used for character input, and should therefore be included by stream classes concerned with character input. The user can provide methods for these generic functions specialized on the user-defined class. Methods for other generic functions must be provided by the user. There is an example in “Defining a new stream class” on page 400. |

**See also**

fundamental-character-stream  
fundamental-input-stream  
stream-clear-input  
stream-listen  
stream-peek-char  
stream-read-char  
stream-read-char-no-hang  
stream-read-line
A class that should be included in stream classes for character output.

Stream

stream-read-sequence
stream-unread-char

fundamental-character-output-stream

Summary
A class that should be included in stream classes for character output.

Package
stream

Superclasses
fundamental-character-stream
fundamental-output-stream

Subclasses
None.

Description
The class fundamental-character-output-stream provides default methods for generic functions used for character output, and should therefore be included by stream classes concerned with character output. The user can provide methods for these generic functions specialized on the user-defined class. Methods for other generic functions must be provided by the user.

There is an example in “Defining a new stream class” on page 400.

See also
fundamental-character-stream
fundamental-input-stream
stream-clear-output
stream-finish-output
stream-force-output
stream-start-line-p
stream-terpri
stream-line-column
stream-write-char
stream-write-sequence
stream-write-string
fundamental-character-stream

**Class**

| Summary | A class whose inclusion provides a method for `stream-element-type` that returns `character`. |
| Package | `stream` |
| Superclasses | `fundamental-stream` |
| Subclasses | `fundamental-character-input-stream`  
              `fundamental-character-output-stream` |
| Description | The class `fundamental-character-stream` is a superclass for character streams. Its inclusion provides a method for the generic function `stream-element-type` that returns the symbol `character`. |
| See also | `fundamental-character-input-stream`  
          `fundamental-character-output-stream`  
          `fundamental-stream`  
          `stream-element-type` |

fundamental-input-stream

**Class**

| Summary | A class whose inclusion causes `input-stream-p` to return `t`. |
| Package | `stream` |
| Superclasses | `fundamental-stream` |
| Subclasses | `fundamental-binary-input-stream`  
              `fundamental-character-input-stream` |
| Description | The class `fundamental-input-stream` is a superclass to the binary and character input classes. Its inclusion causes the generic function `input-stream-p` to return `t`. |
See also  
- `fundamental-binary-input-stream`
- `fundamental-character-input-stream`
- `fundamental-stream`
- `input-stream-p`

---

**fundamental-output-stream**

**Class**

**Summary**  
A class whose inclusion causes `output-stream-p` to return `t`.

**Package**  
`stream`

**Superclasses**  
`fundamental-stream`

**Subclasses**  
- `fundamental-binary-output-stream`
- `fundamental-character-output-stream`

**Description**  
The class `fundamental-output-stream` is a superclass to the binary and character output classes. Its inclusion causes the generic function `output-stream-p` to return `t`.

See also  
- `fundamental-binary-output-stream`
- `fundamental-character-output-stream`
- `fundamental-stream`
- `input-stream-p`

---

**fundamental-stream**

**Class**

**Summary**  
A class whose inclusion causes `streamp` to return `t`.

**Package**  
`stream`

**Superclasses**  
- `standard-object`
- `stream`
<table>
<thead>
<tr>
<th>Subclasses</th>
<th>fundamental-binary-stream</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fundamental-character-stream</td>
</tr>
<tr>
<td></td>
<td>fundamental-input-stream</td>
</tr>
<tr>
<td></td>
<td>fundamental-output-stream</td>
</tr>
</tbody>
</table>

| Description       | The class **fundamental-stream** is a superclass to the fundamental input, output, character and binary streams. Its inclusion causes `streamp` to return t. |

| See also          | close                               |
|                   | fundamental-binary-stream           |
|                   | fundamental-character-stream        |
|                   | fundamental-input-stream            |
|                   | fundamental-output-stream           |
|                   | open-stream-p                        |

---

### stream-advance-to-column

**Generic Function**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Writes the required number of blank spaces to ensure that the next character will be written in a given column.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>stream</td>
</tr>
<tr>
<td>Signature</td>
<td><code>stream-advance-to-column stream column =&gt; result</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>stream</code> A stream.</td>
</tr>
<tr>
<td></td>
<td><code>column</code> An integer.</td>
</tr>
<tr>
<td>Values</td>
<td><code>result</code> A boolean.</td>
</tr>
</tbody>
</table>

| Description        | The generic function `stream-advance-to-column` writes enough blank spaces to `stream` to ensure that the next character is written at `column`. The generic function returns `t` if the operation is successful, or `nil` if it is not supported for this stream. |
This function is intended for use by print and format -t. The default method uses stream-line-column and repeated calls to stream-write-char with a \#\Space character, and returns nil if stream-line-column returns nil.

See also stream-line-column

**stream-check-eof-no-hang**

*Generic Function*

Summary Determines whether a stream is at end of file.

Package stream

Signature stream-check-eof-no-hang stream => result

Arguments stream An input stream.

Values result nil or :eof.

Description The generic function stream-check-eof-no-hang determines if the data source of the stream is at end of file, without hanging.

stream should be an instance of a subclass of buffered-stream.

result is :eof if stream is at end of file and nil otherwise.

There is a built-in method specialized on buffered-stream which returns :eof in all cases.

See also buffered-stream

**stream-clear-input**

*Generic Function*

Summary Implements clear-input.
stream-clear-input

**Package**: stream

**Signature**: stream-clear-input stream => nil

**Arguments**: stream A stream.

**Values**: nil

**Description**: The generic function stream-clear-input implements clear-input. The default method is defined on fundamental-input-stream and does nothing.

**See also**: fundamental-input-stream

---

stream-clear-output

**Generic Function**

**Summary**: Implements clear-output.

**Package**: stream

**Signature**: stream-clear-output stream => nil

**Arguments**: stream A stream.

**Values**: nil

**Description**: The generic function stream-clear-output implements clear-output. The default method is on fundamental-output-stream and does nothing.

There is an example in “Stream output” on page 402.

**See also**: fundamental-output-stream
stream-file-position

Summary
Returns or changes the current position within a stream.

Package
stream

Signature
stream-file-position stream => position

Signature
(setf stream-file-position) position-spec stream => success-p

Arguments
stream A stream.
position-spec A file position designator.

Values
position A file position or nil.
success-p A generalized boolean.

Description
The generic function stream-file-position implements file-position.

stream-file-position is called when file-position is called with one argument.

(setf stream:stream-file-position) is called when file-position is called with two arguments.

The return value is returned by file-position. For the setf function, this is a slight anomaly because setf functions normally return the new value. However in this case it should return the success-p value mandated by the ANSI Common Lisp standard.

The default methods specialized on stream return nil.

stream-fill-buffer

Summary
Fills the stream buffer.

Package
stream
stream-fill-buffer stream => result

Arguments
stream An input stream.

Values
result A generalized boolean.

Description
The generic function stream-fill-buffer is called by the reading functions to fill an empty stream buffer from the underlying data source.

stream should be an instance of a subclass of buffered-stream.

stream-fill-buffer should block until some data is available or return false at end of file. If data is available, it should place it in a buffer, set the stream's input buffer, index and limit appropriately and return a true value. The existing stream buffer can be reused if desired but the index and limit must be updated. The buffer must be of type simple-string, whose element type matches that given when the stream was constructed.

There is a built-in method specialized on buffered-stream which usually suffices. It calls stream-read-buffer with the whole buffer and returns false if this call returns 0. If not, the input index is set to 0 and the input limit is set to the value returned by stream-read-buffer.

See also
buffered-stream
stream-read-buffer

stream-finish-output

Generic Function

Summary
Implements finish-output.

Package
stream

Signature
stream-finish-output stream => nil
The STREAM Package

Arguments

stream

A stream.

Values

nil

Description

The generic function stream-finish-output implements finish-output. The default method is on fundamental-output-stream and does nothing.

There is an example in “Stream output” on page 402.

See also

fundamental-output-stream

stream-flush-buffer

Generic Function

Summary

Flushes a stream’s buffer.

Package

stream

Signature

stream-flush-buffer stream => result

Arguments

stream

An output stream.

Values

result

A generalized boolean.

Description

The generic function stream-flush-buffer is called by the writing functions to flush a stream buffer to the underlying data sink.

stream should be an instance of a subclass of buffered-stream.

Before returning, stream-flush-buffer must set the output index of stream so that more characters can be written to the buffer. If desired, the output buffer and limit can be set too.

There is a built-in method specialized on buffered-stream which usually suffices. It calls stream-write-buffer with
the currently active part of the stream’s output buffer and sets
the output index to 0.

result is true if the buffer was flushed.

See also buffered-stream
        stream-write-buffer

---

stream-force-output  

Generic Function

Summary  Implements force-output.

Package  stream

Signature  stream-force-output stream => nil

Arguments  stream  A stream.

Values  nil

Description  The generic function stream-force-output implements force-output. The default
method is on fundamental-output-stream and does nothing.

There is an example in “Stream output” on page 402.

See also  fundamental-output-stream

---

stream-fresh-line  

Generic Function

Summary  Used by fresh-line to start a new line on a given stream.

Package  stream

Signature  stream-fresh-line stream => bool
Arguments  

stream  
A stream.

Values  

bool  
A generalized boolean.

Description  
The generic function `stream-fresh-line` is used by `fresh-line` to start a new line on a stream. The default method uses `stream-start-line-p` and `stream-terpri`. The result value is `t` if a new line is output successfully.

See also  

`stream-start-line-p`  
`stream-terpri`

**stream-line-column**  

*Generic Function*

Summary  
Returns the column number where the next character will be written.

Package  
stream

Signature  

stream-line-column stream => column

Arguments  

stream  
A stream.

Values  

column  
An integer.

Description  
The generic function `stream-line-column` returns the column number where the next character will be written from `stream`, or `nil` if this is not meaningful for the stream. This function is used in the implementation of `print` and the `format` `-t` directive. A method for this function must be defined for every character output stream class that is defined, although at its simplest it may be defined to always return `nil`.

See also  

`fundamental-character-output-stream`  
`stream-start-line-p`
stream-listen  

**Generic Function**

**Summary**
A function used by `listen` that returns `t` if there is input available.

**Package**
`stream`

**Signature**
`stream-listen stream => result`

**Arguments**
- `stream` A stream.

**Values**
- `result` A generalized boolean.

**Description**
The generic function `stream-listen` is called to determine if there is data immediately available on the stream `stream`, without hanging.

`result` should be true if there is input, and `nil` otherwise (including at end of file).

This method must be implemented for subclasses of `buffered-stream` that handle input.

There is a built-in primary method specialized on `buffered-stream` which returns `nil`. There is a built-in `:around` method specialized on `buffered-stream` which checks for input in the buffer and calls the next method if the buffer is empty. Thus a primary method specialized on a subclass of `buffered-stream` need only check the underlying data source.

The built-in method on `fundamental-input-stream` uses `stream-read-char-no-hang` and `stream-unread-char`. Most streams should define their own method as this is usually trivial and more efficient than the method provided.

**See also**
- `buffered-stream`
- `stream-read-char-no-hang`
- `stream-unread-char`
stream-output-width

Generic Function

Summary
Used by the pretty printer to determine the output width when *print-right-margin* is nil.

Package
stream

Signature
stream-output-width stream => result

Arguments
stream A stream.

Values
result An integer or nil.

Description
The generic function stream-output-width is used by the pretty printer to determine the output width when *print-right-margin* is nil. It returns result, the integer width of stream in units of ems, or nil if the width is not known. The default method provided by fundamental-stream returns nil.

See also
fundamental-stream

stream-peek-char

Generic Function

Summary
A generic function used by peek-char that returns a character on a given stream without removing it from the stream buffer.

Package
stream

Signature
stream-peek-char stream => result

Arguments
stream A stream.

Values
result A character or :EOF symbol.
The generic function stream-peek-char is used to implement peek-char, and corresponds to a peek-type of nil. The default method reads a character from the stream without removing it from the stream buffer, by using stream-read-char and stream-unread-char.

See also

stream-listen
stream-read-char
stream-unread-char

---

### stream-read-buffer

**Generic Function**

**Summary**
Reads data into the stream buffer.

**Package**
stream

**Signature**
stream-read-buffer stream buffer start end => result

**Arguments**
- stream: An input stream.
- buffer: A stream buffer.
- start, end: Bounding indexes for a subsequence of buffer.

**Values**
- result: A non-negative integer.

**Description**
The generic function stream-read-buffer is called by stream-fill-buffer to place characters into the region of the buffer buffer bounded by start and end.

stream should be an instance of a subclass of buffered-stream.

stream-read-buffer should block until some data is available. result should be the number of characters actually placed in the buffer (0 if at end of file). This method must be
implemented for subclasses of $\text{buffered-stream}$ that handle input.

See also $\text{buffered-stream}$
$\text{stream-fill-buffer}$

$\text{stream-read-byte}$

$\text{Summary}$ A generic function used by $\text{read-byte}$ to read an integer or $\text{:eof}$ symbol from a binary stream.

$\text{Package}$ stream

$\text{Signature}$ $\text{stream-read-byte} \ stream \Rightarrow \ result$

$\text{Arguments}$ $\text{stream}$ An input stream.

$\text{Values}$ $\text{result}$ An integer or $\text{:eof}$.

$\text{Description}$ The generic function $\text{stream-read-byte}$ is used by $\text{read-byte}$, and returns either an integer read from the binary stream specified by $\text{stream}$, or the keyword $\text{:eof}$.

A method must be implemented for all binary subclasses of $\text{buffered-stream}$ that handle input. A typical implementation will call $\text{stream-read-char}$ and convert the character to an integer using $\text{char-code}$.

A method should be defined for a subclass of $\text{fundamental-binary-input-stream}$.

See also $\text{buffered-stream}$
$\text{fundamental-binary-input-stream}$
$\text{fundamental-binary-stream}$
$\text{stream-read-char}$
**stream-read-char**

*Generic Function*

**Summary**
Read one character from a stream.

**Package**
*stream*

**Signature**
`stream-read-char stream => character`

**Arguments**
`stream` An input stream.

**Values**
`character` A character or the `:EOF` symbol.

**Description**
The generic function `stream-read-char` reads one item from `stream`. The item read is either a character or the end of file symbol `:EOF` if the stream is at the end of a file. Every subclass of `fundamental-character-input-stream` must define a method for this function.

**See also**
`fundamental-character-input-stream`
`stream-unread-char`

---

**stream-read-char-no-hang**

*Generic Function*

**Summary**
Returns either a character from the stream, an `:eof` if the end-of-file is reached, or `nil` if no input is currently available.

**Package**
*stream*

**Signature**
`stream-read-char-no-hang stream => result`

**Arguments**
`stream` An input stream.

**Values**
`result` Either a character, an `:EOF` symbol, or `nil`.

**Description**
The generic function `stream-read-char-no-hang` implements `read-char-no-hang`. It returns either a character read
from the stream, or :eof if end-of-file is reached, or nil if no input is available. The default method provided by fundamental-character-input-stream simply calls stream-read-char which is sufficient for file streams, but interactive streams should define their own method.

See also  
fundamental-character-input-stream  
stream-read-char

**stream-read-line**  
*Generic Function*

**Summary**  Returns a string read from a stream.

**Package**  stream

**Signature**  
stream-read-line stream => result terminated

**Arguments**  
stream  An input stream.

**Values**  
result  A string or :eof.  
terminated  A boolean.

**Description**  The generic function stream-read-line reads a line of characters from stream and returns this line as a string. If the string is terminated by an end-of-file instead of a newline then terminated is t.

The default method uses repeated calls to stream-read-char, and uses stream-element-type to determine the element-type of its result.

See also  
fundamental-character-input-stream  
stream-element-type  
stream-read-char

1702
stream-read-sequence

Generic Function

Summary
Reads a number of items from a stream into a sequence.

Package
stream

Signature
stream-read-sequence stream sequence start end => index

Arguments
stream A stream.
sequence A sequence.
start An integer.
end An integer.

Values
index An integer.

Description
The generic function stream-read-sequence reads from stream into sequence. Elements from the start of sequence are replaced by elements from stream until end in sequence or the end-of-file in stream is reached. The index of the first element in sequence that is not replaced is returned.

A default method is provided by fundamental-character-input-stream which makes repeated calls to stream-read-char and uses (setf elt) to insert characters into sequence. A default method is provided by fundamental-binary-input-stream that makes repeated calls to stream-read-byte and also uses (setf elt) to insert bytes into sequence. Note that this may lead to error if the sequence is of inappropriate type.

See also
fundamental-binary-input-stream
fundamental-character-input-stream
stream-read-byte
stream-read-char
**stream-read-timeout**  
*Generic Function*

**Summary**  
Accesses the read-timeout property of a socket stream.

**Package**  
*stream*

**Signature**  
`stream-read-timeout stream => timeout`

**Arguments**  
`stream`  
A socket stream.

**Values**  
`timeout`  
A positive number or nil.

**Description**  
The generic function `stream-read-timeout` reads the current `read-timeout` of an instance of `comm:socket-stream`.  

`(setf stream-read-timeout)` sets the read-timeout of an instance of `comm:socket-stream`.

**See also**  
socket-stream  
open-tcp-stream

---

**stream-start-line-p**  
*Generic Function*

**Summary**  
A generic function that returns t if the stream is positioned at the beginning of a line.

**Package**  
*stream*

**Signature**  
`stream-start-line-p stream => result`

**Arguments**  
`stream`  
A stream.

**Values**  
`result`  
A boolean.

**Description**  
The generic function `stream-start-line-p` returns t if `stream` is positioned at the beginning of a line, and nil other-
wise. It is permissible to define a method that always returns nil.

Note that although a value of 0 from stream-line-column also indicates the beginning of a line, there are cases where stream-start-line-p can be meaningfully implemented and stream-line-column cannot. For example, for a window using variable-width characters the column number is not very meaningful, whereas the beginning of a line has a clear meaning.

The default method for stream-start-line-p on class fundamental-character-output-stream uses stream-line-column. Therefore, if this is defined to return nil, a method should be provided for either stream-start-line-p or stream-fresh-line.

See also fundamental-character-output-stream stream-fresh-line stream-line-column

stream-terpri

Generic Function

Summary Writes an end of line to a stream.

Package stream

Signature stream-terpri stream => nil

Arguments stream A stream.

Values nil

Description The generic function stream-terpri writes an end of line to stream, as for terpri. The default method for stream-terpri is (stream-write-char stream #\Newline).
See also stream-write-char

**stream-unread-char**

*Generic Function*

**Summary**
Undoes the last call to stream-read-char.

**Package**
stream

**Signature**
stream-unread-char stream character => nil

**Arguments**
stream A stream.
character A character.

**Values**
nil

**Description**
The generic function stream-unread-char undoes the last call to stream-read-char, as in unread-char. Every subclass of fundamental-character-input-stream must define a method for this function.

See also fundamental-character-input-stream

**stream-write-buffer**

*Generic Function*

**Summary**
Writes a part of stream's buffer.

**Package**
stream

**Signature**
stream-write-buffer stream buffer start end

**Arguments**
stream An output stream.
buffer A stream buffer.
start, end Bounding indexes for a subsequence of buffer.
The generic function `stream-write-buffer` is called by `stream-flush-buffer` to write the region of the buffer bounded by `start` and `end` to the stream’s underlying data sink.

`stream` should be an instance of a subclass of `buffered-stream`.

This method must be implemented for subclasses of `buffered-stream` that handle output.

See also `buffered-stream`  
`stream-flush-buffer`

---

**stream-write-byte**  
**Generic Function**

**Summary**  
A generic function used by `write-byte` to write an integer to a binary stream.

**Package**  
`stream`

**Signature**  
`stream-write-byte stream integer => result`

**Arguments**  
- `stream`  
A stream.
- `integer`  
An integer.

**Values**  
- `result`  
An integer.

**Description**  
The generic function `stream-write-byte` is used by `write-byte`, and writes the integer `integer` to the binary stream specified by `stream`.

A method must be implemented for all binary subclasses of `buffered-stream` that handle output. A typical implementation will convert the integer to a character using `code-char` and call `stream-write-char`.
A method should be defined for all subclasses of `fundamental-binary-output-stream`.

See also `buffered-stream`
`fundamental-binary-output-stream`
`fundamental-binary-stream`
`stream-write-char`

---

### stream-write-char

**Generic Function**

**Summary**
Writes a character to a specified stream.

**Package**
`stream`

**Signature**
`stream-write-char stream character => character`

**Arguments**
- `stream` A stream.
- `character` A character.

**Values**
- `character` A character.

**Description**
The generic function `stream-write-char` writes `character` to `stream`. Every subclass of `fundamental-character-output-stream` must have a method defined for this function.

There is an example in “Stream output” on page 402.

See also `fundamental-character-output-stream`

---

### stream-write-sequence

**Generic Function**

**Summary**
Writes a subsequence of a sequence to a stream.

**Package**
`stream`
Signature  
\textit{stream-write-sequence} stream sequence start end => result

Arguments  
\textit{stream} A stream.  
\textit{sequence} A sequence.  
\textit{start} An integer.  
\textit{end} An integer.

Values  
\textit{result} A sequence.

Description  
The generic function \textit{stream-write-sequence} is used by \textit{write-sequence} to write a subsequence of \textit{sequence} delimited by \textit{start} and \textit{end} to \textit{stream}.

A default method is provided by \textit{fundamental-character-output-stream} that tests each element of \textit{sequence} in turn, and then uses \textit{stream-write-char} or produces an error. A default method is provided by \textit{fundamental-binary-output-stream} that tests each element of \textit{sequence} in turn, and then uses \textit{stream-write-byte} or produces an error.

See also  
\begin{itemize}  
\item \textit{fundamental-binary-output-stream}  
\item \textit{fundamental-character-output-stream}  
\item \textit{stream-read-sequence}  
\item \textit{stream-write-byte}  
\item \textit{stream-write-char}  
\end{itemize}

\textbf{stream-write-string}  
\textit{Generic Function}

Summary  
Used by \textit{write-string} to write a string to a character output stream.

Package  
\textit{stream}

Signature  
\textit{stream-write-string} stream string &optional start end => result

1709
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>A stream.</td>
</tr>
<tr>
<td>string</td>
<td>A string.</td>
</tr>
<tr>
<td>start</td>
<td>An integer.</td>
</tr>
<tr>
<td>end</td>
<td>An integer.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>A string.</td>
</tr>
</tbody>
</table>

Description

The generic function `stream-write-string` is used by `write-string` to write `string` to `stream`. The string can, optionally, be delimited by `start` and `end`.

The default method provided by `fundamental-character-output-stream` uses repeated calls to `stream-write-char`.

There is an example in “Stream output” on page 402.

See also

- `fundamental-character-output-stream`
- `stream-write-char`

### with-stream-input-buffer

**Macro**

Summary

Allows access to the input buffer.

Package

`stream`

Signature

```
with-stream-input-buffer (buffer index limit) stream &body body => result
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer, index, limit</td>
<td>Variables.</td>
</tr>
<tr>
<td>stream</td>
<td>An input stream.</td>
</tr>
<tr>
<td>body</td>
<td>Code.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>The value returned by <code>body</code>.</td>
</tr>
</tbody>
</table>
The macro `with-stream-input-buffer` allows access to the state of the input buffer for the given buffered stream.

`stream` should be an instance of a subclass of `buffered-stream`.

Within the code body, the variables `buffer`, `index` and `limit` are bound to the buffer of `stream`, its current index and the limit of the buffer. Setting `buffer`, `index` or `limit` will change the values in the stream `stream` but note that other changes to these values (for example, by calling other stream functions) will not affect the values bound within the macro. See the example for a typical use which shows how this restriction can be handled.

The buffer is always of type `simple-string`. The `stream-element-type` of `stream` depends on how it was constructed.

The index is the position of the next element to be read from the buffer and the limit is the position of the element after the end of the buffer. Therefore there is no data in the buffer when `index` is greater than or equal to `length`.

This example function returns a string with exactly four characters read from a buffered stream. If `end-of-file` is reached before four characters have been read, it returns `nil`. 

---

**Example**

This example function returns a string with exactly four characters read from a buffered stream. If `end-of-file` is reached before four characters have been read, it returns `nil`. 

---

**Description**

The macro `with-stream-input-buffer` allows access to the state of the input buffer for the given buffered stream.

`stream` should be an instance of a subclass of `buffered-stream`.

Within the code body, the variables `buffer`, `index` and `limit` are bound to the buffer of `stream`, its current index and the limit of the buffer. Setting `buffer`, `index` or `limit` will change the values in the stream `stream` but note that other changes to these values (for example, by calling other stream functions) will not affect the values bound within the macro. See the example for a typical use which shows how this restriction can be handled.

The buffer is always of type `simple-string`. The `stream-element-type` of `stream` depends on how it was constructed.

The index is the position of the next element to be read from the buffer and the limit is the position of the element after the end of the buffer. Therefore there is no data in the buffer when `index` is greater than or equal to `length`.
(defun read-4-chars (stream)
  (declare (type stream:buffered-stream stream))
  (let ((res (make-string 4))
        (elt 0))
    ;; Outer loop handles buffer filling.
    (loop)
    ;; Inner loop handles buffer scanning.
    (loop (stream:with-stream-input-buffer (buf ind lim) stream
      (when (>= ind lim)
        ;; End of buffer: try to refill.
        (return))
      (setf (schar res elt) (schar buf ind))
      (incf elt)
      (incf ind)
      (when (= elt 4)
        (return-from read-4-chars res))))
    (unless (stream:stream-fill-buffer stream)
      (return-from read-4-chars nil))))

See also
buffered-stream
with-stream-output-buffer

with-stream-output-buffer

Macro

Summary
Allows access to the output buffer.

Package
stream

Signature
with-stream-output-buffer (buffer index limit) stream &body
body => result

Arguments
buffer, index, limit

Variables
stream An output stream
body Code

Values
result The value returned by body.
The macro `with-stream-output-buffer` allows access to the state of the output buffer for the given buffered stream. `stream` should be an instance of a subclass of `buffered-stream`.

Within the code body, the variable names `buffer`, `index` and `limit` are bound to the buffer of `stream`, its current index and the limit of the buffer. Setting `buffer`, `index` or `limit` will change the values in the stream `stream` but note that other changes to these values (for example, by calling other stream functions) will not affect the values bound within the macro. See the example for a typical use which shows how this restriction can be handled.

The buffers are always of type `simple-string`. The `stream-element-type` of `stream` depends on how the stream was constructed.

The index is the position of the next free element in the buffer and the limit is the position of the element after the end of the buffer. Therefore the buffer is full when `index` is greater than or equal to `length`.

**Example**

This example function writes a four character string to a buffered stream.
(defun write-4-chars (stream string)
  (declare (type stream:buffered-stream stream))
  (let ((elt 0))
    ;; Outer loop handles buffer flushing.
    (loop
      ;; Inner loop handles buffer updating.
      (loop (stream:with-stream-output-buffer (buf ind lim) stream
        (when (>= ind lim)
          ;; Buffer full: try to flush.
          (return))
        (setf (schar buf ind) (schar string elt))
        (incf elt)
        (incf ind)
        (when (= elt 4)
          (return-from write-4-chars))))
    (stream:stream-flush-buffer stream))))

See also
buffered-stream
with-stream-input-buffer
This chapter describes symbols available in the `SYSTEM` package.
Various uses of the symbols documented here are discussed throughout this manual.

**allocated-in-its-own-segment-p**

*Function*

Summary 64-bit LispWorks only: Returns if the object is allocated in its own segment.

Package `system`

Signature `allocated-in-its-own-segment-p object => result`

Arguments `object` Any object.

Values `result` A boolean.

Description The function `allocated-in-its-own-segment-p` returns true if `object` is allocated in its own own segment and false otherwise.
Notes
An object is allocated in its own segment if it is "very large". Currently that means larger than 64 MB for the ordinary 64-bit GC, or larger than 1 MB for the Mobile GC.

allocated-in-its-own-segment-p is intended to help to decide whether to call the functions that are useful only for such objects (make-object-permanent and release-object-and-nullify).

See also
make-object-permanent
release-object-and-nullify
"Mobile GC technical details" on page 154

apply-with-allocation-in-gen-num

Function

Summary
Allows control over which generation objects are allocated in, in 64-bit LispWorks.

Package
system

Signature
apply-with-allocation-in-gen-num what gen-num func &rest args => results

Arguments
what One of the keywords :cons, :symbol, :function, :non-pointer and :other.

gen-num An integer in the inclusive range [0,7], or nil.

func A function designator.

args The arguments passed to func.

Values
results The values returned from the call to func with args.

Description
The function apply-with-allocation-in-gen-num applies the function func to args such that objects of allocation type
what are allocated in generation gen-num, in 64-bit LispWorks.

See also the keyword :allocation to make-array, which catches the most common cases.

It is probably quite rare that it is useful to use this function, unless the function allocates a lot, and you are certain that every object that is allocated of the allocation type is long-lived, which is normally difficult to tell.

Notes
1. Allocation of interned symbols is controlled separately by *symbol-alloc-gen-num*.
2. In 32-bit LispWorks the argument what is ignored and the effect is like that of the macro allocation-in-gen-num.
3. In the Mobile GC, gen-num must be 0, 1 or 2.

See also
allocation-in-gen-num
make-array
*symbol-alloc-gen-num*

approaching-memory-limit

Summary
The class of conditions signalled when 32-bit LispWorks approaches its memory limit.

Package system

Superclasses storage-condition

Subclasses None.

Description The condition class approaching-memory-limit is used for signalling an error when 32-bit LispWorks approaches its memory limit.
Notes
appaching-memory-limit is not relevant to 64-bit Lisp-Works.

See also
“Approaching the memory limit” on page 142
set-approaching-memory-limit-callback

atomic-decf atomic-incf

Macros

Summary
Like incf and decf, but does the operation atomically.

Package
system

Signatures
atomic-decf place &optional delta => new-value
atomic-incf place &optional delta => new-value

Arguments
place One of the specific set of places defined for
low level atomic operations.

delta A number, default value 1.

Values
new-value A number

Description
The macro atomic-decf is like decf and atomic-incf is like
incf, except that they are guaranteed atomic for a suitable
place.

place must be one of the places described in “Low level
atomic operations” on page 299, or expand to one of them.

Notes
Unlike atomic-fixnum-decf and atomic-fixnum-incf,
these macros can deal with any number.

See also
atomic-fixnum-decf
atomic-fixnum-incf
low-level-atomic-place-p
atomic-exchange

Macro

Summary
Atomically exchange a place value with a new value, returning the old value.

Package
system

Signature
atomic-exchange place new-value => old-value

Arguments
place One of the specific set of places defined for low level atomic operations.
new-value An object.

Values
old-value An object.

Description
The macro atomic-exchange exchanges the value in place with new-value, returning the old-value. The operation is guaranteed to be atomic.
place must be one of the places described in “Low level atomic operations” on page 299, or expand to one of them.

See also
compare-and-swap
low-level-atomic-place-p

atomic-fixnum-decf
atomic-fixnum-incf

Macros

Summary
Like defc and incf, but does the operation atomically.

Package
system

Signature
atomic-fixnum-decf place &optional fixnum-delta => new-value
atomic-fixnum-incf place &optional fixnum-delta => new-value
Arguments

place
One of the specific set of places defined for low level atomic operations.

fixnum-delta
A fixnum, default value 1

Values

new-value
A fixnum.

Description

The macro atomic-fixnum-decf is like decf (for fixnums only) and atomic-fixnum-incf is like incf (for fixnums only), except that they are guaranteed atomic for a suitable place.

place must be one of the places described in “Low level atomic operations” on page 299, or expand to one of them.

Both the value in the place and fixnum-delta must be fixnums. The arithmetic is done without checking for overflow.

See also

atomic-decf
atomic-incf
low-level-atomic-place-p

atomic-pop

Macro

Summary

Like pop, but does the operation atomically.

Package

system

Signature

atomic-pop place => element

Arguments

place
One of the specific set of places defined for low level atomic operations.

Values

element
An object.

Description

The macro atomic-pop is the same as cl:pop, but is guaranteed atomic for a suitable place.
place must be one of the places described in “Low level atomic operations” on page 299, or expand to one of them.

See also  atomic-push
          low-level-atomic-place-p

**atomic-push**  
*Macro*

**Summary**  
Like push, but does the operation atomically.

**Package**  
system

**Signature**  
amatomic-push new-value place => new-place-value

**Arguments**  
new-value  
An object.

place  
One of the specific set of places defined for low level atomic operations.

**Values**  
new-place-value  
A list (the new value of place).

**Description**  
The macro atomic-push is the same as cl:push, but is guaranteed atomic for a suitable place.

place must be one of the places described in “Low level atomic operations” on page 299, or expand to one of them.

**Notes**  
in many cases the natural inverse of push is delete, but there is no way to do delete atomically, except by using a separate lock, which must also be held while doing the push.

See also  atomic-pop
          low-level-atomic-place-p
**augmented-string**

**simple-augmented-string**

*Types*

**Summary**

Deprecated synonyms for `text-string` and `simple-text-string`.

**Package**

`system`

**Signature**

`augmented-string length`

`simple-augmented-string length`

**Arguments**

`length` The length of the string (or *, meaning any).

**Description**

The types `augmented-string` and `simple-augmented-string` are deprecated synonyms for `text-string` and `simple-text-string`.

**See also**

`text-string`

`simple-text-string`

---

**augmented-string-p**

**simple-augmented-string-p**

*Functions*

**Summary**

Deprecated synonyms for `text-string-p` and `simple-text-string-p`.

**Package**

`system`

**Signature**

`augmented-string-p object => result`

`simple-augmented-string-p object => result`

**Arguments**

`object` A Lisp object.

**Values**

`result` A boolean.
<table>
<thead>
<tr>
<th>Description</th>
<th>The functions \texttt{augmented-string-p} and \texttt{simple-augmented-string-p} are deprecated synonyms for \texttt{text-string-p} and \texttt{simple-text-string-p}.</th>
</tr>
</thead>
<tbody>
<tr>
<td>See also</td>
<td>\texttt{text-string-p}</td>
</tr>
<tr>
<td></td>
<td>\texttt{simple-text-string-p}</td>
</tr>
</tbody>
</table>

\*binary-file-type*  

<table>
<thead>
<tr>
<th>Summary</th>
<th>The default file type of binary files.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>system</td>
</tr>
<tr>
<td>Initial value</td>
<td>The initial value of *binary-file-type* depends on the host CPU and the LispWorks implementation. See “Naming conventions for FASL files” on page 648.</td>
</tr>
<tr>
<td>Description</td>
<td>The variable *binary-file-type* is the file type that \texttt{load} and \texttt{require} recognize as a binary (FASL) file (in addition to any additional file types in *binary-file-types*). Normally you should not set *binary-file-type*. If you need to load files with another type, push that type on to *binary-file-types*.</td>
</tr>
<tr>
<td>See also</td>
<td>*binary-file-types*</td>
</tr>
<tr>
<td></td>
<td>\texttt{load-data-file}</td>
</tr>
</tbody>
</table>

\*binary-file-types*  

<table>
<thead>
<tr>
<th>Summary</th>
<th>A list of file types that are loaded as binary files.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>system</td>
</tr>
<tr>
<td>Initial value</td>
<td>nil</td>
</tr>
</tbody>
</table>

1723
The variable \texttt{*binary-file-types*} contains a list of strings naming file types which \texttt{load} and \texttt{require} recognize as binary (FASL) files. FASL files are the output of \texttt{compile-file}, \texttt{dump-forms-to-file} or \texttt{with-output-to-fasl-file}.

You need to add a type to this list if you want load such files but they have an extension which is different from the default (the value of \texttt{*binary-file-type*}).

See also \texttt{*binary-file-type*}

\section*{call-system \ Function}

\textbf{Package} \hspace{2em} \texttt{system}

\textbf{Signature} \hspace{2em} \texttt{call-system command \&key current-directory wait shell-type} \Rightarrow status, signal-number

\textbf{Arguments} \hspace{2em}
- \texttt{command} \hspace{2em} A string, a list of strings, a simple-vector of strings, or \texttt{nil}.
- \texttt{current-directory} \hspace{2em} A string. Implemented only on Microsoft Windows.
- \texttt{wait} \hspace{2em} A boolean.
- \texttt{shell-type} \hspace{2em} A string or \texttt{nil}.

\textbf{Values} \hspace{2em}
- \texttt{status} \hspace{2em} The exit status of the invoked shell or process.
- \texttt{signal-number} \hspace{2em} On Unix-like systems, if the process was terminated by a signal this is the signal number, otherwise \texttt{nil}. Always \texttt{nil} on Microsoft Windows.

\textbf{Description} \hspace{2em} \texttt{call-system} allows executables and DOS or Unix shell commands to be called from Lisp code as a separate OS process.
The output goes to standard output, as the operating system sees it. (This normally means *terminal-io* in LispWorks.)

If command is a string then it is passed to the shell as the command to run, using the -c option, without any other arguments. The type of shell to run is determined by shell-type as described below. Note that for typical Unix shells, the string command may contain multiple commands separated by `;` (semicolon).

If command is a list then it becomes the argv of a command to run directly, without invoking a shell. The first element is the command to run directly and the other elements are passed as arguments on the command line (that is, element 0 has its name in argv[0] in C, and so on).

If command is a simple vector of strings, the element at index 0 is the command to run directly, without invoking a shell. The other elements are the complete set of arguments seen by the command (that is, element 1 becomes argv[0] in C, and so on).

If command is nil, then the shell is run.

On Microsoft Windows, if command is a string, LispWorks hides the first window of the execution of the command, because that is the console that cmd.exe starts in a DOS window. If the command itself is a console application, you may want to see the console. In this case run the command as a direct command. To do this, pass a list or a vector as described above. Conversely, if you run a console application and do not want to see the console, pass the command as a string.

On Microsoft Windows current-directory is the lpCurrentDirectory argument passed to CreateProcess. If this is not supplied, the pathname-location of the current-pathname is passed.

If wait is true, call-system does not return until the process has exited. The default for wait is t.
On non-Windows platforms, if `shell-type` is a string it specifies the shell. If `shell-type` is `nil` (the default) then the Bourne shell, `/bin/sh`, is used. The C shell may be obtained by passing "/bin/csh".

On supported versions of Microsoft Windows if `shell-type` is `nil` then `cmd.exe` is used.

`call-system` returns the exit status of the process it created. Additionally on Unix-like systems if the process was terminated by a signal then `call-system` returns the number of that signal. For a discussion of these return values see “Interpreting the exit status” on page 462.

**Notes**

If you need to be able to check whether the child process is alive and maybe to kill it, use `open-pipe` with `save-exit-status` `t` (and maybe `:direction :none`) instead of `call-system`, and then use `pipe-exit-status` and maybe `pipe-kill-process`.

**Compatibility notes**

1. The argument `:shell-type` is not implemented in LispWorks for Windows 4.4 and earlier, and `cmd.exe` is not used implicitly.

2. On Microsoft Windows, LispWorks 5.0 and later use `shell-type cmd.exe` by default when `command` is a string. In LispWorks 5.x the user may see a DOS command window in this case, but LispWorks 6.0 and later explicitly hide the DOS window. To call your command directly `command` should be a list, as in the last example below.

**Example**

On Unix:

```lisp
(call-system (format nil "adb ~a < ~a > ~a"
              (namestring a)
              (namestring b)
              (namestring c)))
```

On Microsoft Windows:

```lisp
```
(sys:call-system "sleep 3" :wait t)
(sys:call-system '("notepad" "myfile.txt"))

See also
open-pipe
call-system-showing-output
run-shell-command
"Interpreting the exit status" on page 462

call-system-showing-output

Function

Package system

Signature call-system-showing-output command &key current-directory prefix show-cmd output-stream wait shell-type kill-process-on-abort => status

Arguments

command A string, a list of strings, a simple-vector of strings, or nil.
current-directory A string. Supported only on Microsoft Windows.
prefix A string.
show-cmd A boolean.
output-stream An output stream or nil, t or :tty.
wait A boolean.
shell-type A string. Supported only on non-Windows platforms.
kill-process-on-abort A generalized boolean.

Values

status The exit status of the invoked shell or process.
Description

`call-system-showing-output` is an extension to `call-system` which allows output to be redirected. On non-Windows platforms this means it can be redirected to places other than the shell process from which the LispWorks image was invoked. `call-system-showing-output` therefore allows the user to, for example, invoke a shell command and redirect the output to the current Listener window.

The argument `command` is interpreted as by `call-system`. On Microsoft Windows there is one difference: when `command` is a list or vector and the executable (that is, the first element of the sequence `command`) in `call-system-showing-output` is not a GUI application, LispWorks hides the first window, which is the console that the executable will normally open. Note that for a non-direct command (that is, a string) LispWorks always hides the first window (which is the console) in both `call-system` and `call-system-showing-output`.

`prefix` is a prefix to be printed at the start of any output line. The default value is "; ".

`show-cmd` specifies whether or not the `cmd` invoked will be printed as well as the output for that command. If `t` then `cmd` will be printed. The default value for `show-cmd` is `t`.

`output-stream` specifies where the output will be sent to. If `output-stream` is an output stream, the output is written to it. If `output-stream` is `t`, the output is written to `*standard-output*`. If `output-stream` is `nil`, the output is collected as a string, and returned as a second value from `call-system-showing-output`. If `output-stream` is `:tty`, `call-system-showing-output` behaves like `call-system`. The default value is `*standard-output*`.

If `wait` is true, `call-system-showing-output` does not return until the process has exited. If `nil`, `call-system-showing-output` returns immediately and no output is shown. The default for `wait` is `t`. 

1728
shell-type is a string naming a UNIX shell. The default is "/bin/sh".

If kill-process-on-abort is true, then when call-system-showing-output is aborted the process is killed. The default value of kill-process-on-abort is nil.

call-system-showing-output returns the exit status of the shell invoked to execute the command on non-Windows platforms, or the process created on Microsoft Windows.

Examples

On Linux:

CL-USER 1 > (sys:call-system-showing-output "pwd" :prefix "***")

***pwd
***/amd/xanfs1-cam/u/ldisk/sp/lispsrc/v42/builds
0

CL-USER 2 > (sys:call-system-showing-output "pwd" :prefix "&&&" :show-cmd nil)

&&/amd/xanfs1-cam/u/ldisk/sp/lispsrc/v42/builds
0

On Microsoft Windows:

CL-USER 223 > (sys:call-system-showing-output "cmd /c type hello.txt" :prefix "***")

***cmd /c type hello.txt
***Hi there
0

CL-USER 224 > (sys:call-system-showing-output "cmd /c type hello.txt" :prefix "&&&" :show-cmd nil)

&&Hi there
0

See also
call-system
open-pipe
run-shell-command
cdr-assoc

Summary
A generalized reference for alist elements.

Package
system

Signature
cdr-assoc item alist &key test test-not key => result

setf (cdr-assoc item place &key test test-not key) value => value

Arguments
item An object.
alist An association list.
test A function designator.
test-not A function designator.
key A function designator.
place A setf place containing an association list.
value An object.

Values
result An object (from alist) or nil.

Description
The accessor cdr-assoc provides a generalized reference for elements in an association list. The arguments are all as specified for the Common Lisp function assoc. cdr-assoc and its setf expander read and write the cdr of an element in a manner consistent with the Common Lisp notion of places.

cdr-assoc returns the cdr of the first cons in the alist that matches item (tested using test, test-not and key as for assoc), or nil if no element of alist matches.

Using cdr-assoc with setf modifies the first cons in the value of place that matches item, setting its cdr to value. If no element matches, then it pushes the value of (cons item value) onto the value of place and sets place to this new alist. This is similar to how cl:getf is defined.
When *place* is a globally accessible place that may be read by another thread without synchronization (by a lock or other synchronization mechanism), you need to wrap *alist* by *globally-accessible*. See “Making an object’s contents accessible to other threads” on page 268 for a discussion.

**Example**

```
CL-USER 1 > (defvar *my-alist*
  (list (cons :foo 1)
       (cons :bar 2)))

*MY-ALIST*

CL-USER 2 > (setf (sys:cdr-assoc :bar *my-alist*) 3)
3

CL-USER 3 > *my-alist*
((:FOO . 1) (:BAR . 3))
```

### *check-network-server*

**Variable**

**Summary**

Indicates the presence of a network license.

**Note:** LispWorks for UNIX only.

**Package**

*system*

**Description**

This should always be set to `t` for a site (that is, network) license — the licensing mechanism does not work in any other circumstances. Do not set the variable otherwise, as it overrides any useful diagnostics which may accompany key-file errors. Not applicable to LispWorks for Linux, Windows, x86/x64 Solaris, FreeBSD, AIX or Macintosh.

### coerce-to-gesture-spec

**Function**

**Summary**

Returns a Gesture Spec object.

**Package**

*system*
Signature  

\texttt{coerce-to-gesture-spec \hspace{2pt} object \hspace{2pt} \&optional \hspace{2pt} errorp \hspace{2pt} \Rightarrow \hspace{2pt} gspec}

Arguments  

\texttt{object} \hspace{2pt} A character, keyword, Gesture Spec or string.

\texttt{errorp} \hspace{2pt} A boolean.

Values  

\texttt{gspec} \hspace{2pt} A Gesture Spec object

Description  

The function \texttt{coerce-to-gesture-spec} returns a Gesture Spec object \texttt{gspec} which can be used to represent the keystroke indicated by \texttt{object}.

If \texttt{object} is a Lisp character, then \texttt{gspec}'s data is its \texttt{cl:char-code} and \texttt{gspec}'s modifiers are 0.

If \texttt{object} is a keyword, then it must be one of the known Gesture Spec keywords and becomes \texttt{gspec}'s data. \texttt{gspec}'s modifiers is 0.

If \texttt{object} is a string, then \texttt{coerce-to-gesture-spec} expects it to be a sequence of modifier key names separated by the - character, followed by a single character or a character name as returned by \texttt{name-char} or the name of one of the known Gesture Spec keywords. Then \texttt{gspec} contains the corresponding Gesture Spec keyword or \texttt{char-code} in its \texttt{data}, and the modifier keys are represented in its \texttt{modifiers}.

If \texttt{object} is a Gesture Spec object, it is simply returned.

\texttt{coerce-to-gesture-spec} does not create wild gesture specs.
Examples

(sys:coerce-to-gesture-spec :F10)
=>
#S(SYSTEM::GESTURE-SPEC :DATA :F10 :MODIFIERS 0)

(sys:coerce-to-gesture-spec "Ctrl-C")
=>
#S(SYSTEM::GESTURE-SPEC :DATA 67 :MODIFIERS 2)

(sys:coerce-to-gesture-spec "Shift-F10")
=>
#S(SYSTEM::GESTURE-SPEC :DATA :F10 :MODIFIERS 1)

See also
gesture-spec-control-bit
gesture-spec-data
gesture-spec-modifiers
gesture-spec-p
gesture-spec-to-character
make-gesture-spec
print-pretty-gesture-spec

compare-and-swap

Macro

Summary
Performs a conditional store, atomically.

Package
system

Signature
compare-and-swap place compare new-value => result

Arguments
place One of the specific set of places defined for low level atomic operations.

compare An object.

new-value An object.

Values
result A boolean.
The macro `compare-and-swap` compares the value in `place` with `compare`, and if they are the same (by `eq`), stores the new-value in `place`.

`compare-and-swap` returns non-nil if the store occurred, or `nil` if the store did not occur.

`place` must be one of the places described in “Low level atomic operations” on page 299, or expand to one of them. The operation is guaranteed to be atomic.

See also `atomic-exchange`

`low-level-atomic-place-p`

copy-preferences-from-older-version

Function

Summary
Copies uses preferences.

Package
`system`

Signature
copy-preferences-from-older-version `old-path` `new-path`
&optional `flag-name`

Arguments
`old-path` A preference path.
`new-path` A preference path.
`flag-name` A string.

Description
The function `copy-preferences-from-older-version` copies uses preferences from one part of the registry to another.

`old-path` and `new-path` are the paths of preferences for the old and the new version, corresponding to the paths that were passed to `(setf product-registry-path)`.

`flag-name` is a name of the flag to use to record in the registry that the copy is already done. `flag-name` must be a valid regis-
try value name on Microsoft Windows, and a valid filename on all other platforms. The default value of flag-name is the string "copied-old-preferences".

`copy-preferences-from-older-version` performs several checks:

1. It checks whether it already copied to new-path in the current session, and if so does nothing.
2. It checks whether the flag-name entry exists, and if so it does nothing.
3. It checks whether another call to `copy-preferences-from-older-version` is already executing (in another thread), and if so it just waits for the other call to finish.

Then if all the checks above indicate that copying is still needed, `copy-preferences-from-older-version` copies the values from the tree below old-path to a tree below new-path. It traverses the entire tree below old-path, and checks each key to see if it has any values.

For a key that has values, it checks whether the key exists under new-path, and if the key exists it does not copy any of the values for this key, though it still traverses and maybe copies its subkeys. If the key does not exist under new-path, it creates the key and copies the values.

Because it makes checks before doing any work, `copy-preferences-from-older-version` is an inexpensive call that can be used freely.

See also `product-registry-path` `user-preference`

**count-gen-num-allocation**

*Function*

**Summary**

Returns the amount of allocated data in a generation in 64-bit LispWorks.
The function `count-gen-num-allocation` returns the amount of allocated data in generation `gen-num`. If `include-lower-generations` is non-nil, the returned value `allocation` also includes the data in the younger generations.

Notes

`count-gen-num-allocation` is implemented only in 64-bit LispWorks. It is not relevant to the Memory Management API in 32-bit implementations, where you can use `room-values` instead.

On the Mobile GC, the `gen-num` argument can be 0, 1, 2 or 3 (3 means permanent).

See also `room-values`
start the LispWorks IDE automatically, an error during initialization is handled and displayed in a snapshot debugger after the IDE starts.

If the value of `debug-initialization-errors-in-snapshot` is `nil` LispWorks behaves like LispWorks 5.0 and previous versions. That is, it attempts to enter the command line debugger.

---

**default-eol-style**

**Function**

**Summary**

Provides a default end of line style for a file.

**Package**

`system`

**Signature**

`default-eol-style pathname ef-spec buffer length => new-ef-spec`

**Arguments**

- `pathname` Pathname identifying location of `buffer`.
- `ef-spec` An external format spec.
- `buffer` A buffer whose contents are examined.
- `length` Length (an integer) up to which `buffer` should be examined.

**Values**

`new-ef-spec` A new external format spec created by merging `ef-spec` with the encoding that was found.

**Description**

Merge `ef-spec` with `(:default :eol-style :crlf)` on Microsoft Windows, `(:default :eol-style :lf)` on non-Windows platforms. This is usually used as the last function on its list.

**See also**

`*file-eol-style-detection-algorithm*`
*default-stack-group-list-length*  

**Variable**

Summary  
The size of the stack cache.

Package  
**system**

Initial value  
10

Description  
This variable determines the maximum size of the stack cache.

Process stacks are cached and reused. When a process dies, its stack is put in the stack cache for future reuse if there are currently less than *default-stack-group-list-length* stacks in the cache. Therefore if your application repeatedly creates and discards more than 10 processes you should consider increasing the value of this variable.

Note that stacks are allocated in generation 2, hence a program with a high turnover of processes may need to call (gc-generation t) periodically unless all the stacks of dead processes are reused.

The default stack size is 64 KB on all 32-bit LispWorks x86 platforms.

**See also**  
mark-and-sweep

**define-atomic-modify-macro**  
**Macro**

Summary  
An atomic version of define-modify-macro.

Package  
**system**

Signature  
define-atomic-modify-macro name lambda-list function  
&optional doc-string => name

Arguments  
name  
A symbol.
\textit{lambda-list} \hspace{1em} A \texttt{define-modify-macro} lambda list.

\textit{function} \hspace{1em} A symbol.

\textit{doc-string} \hspace{1em} A string, not evaluated.

\textbf{Values} \hspace{1em} \textit{name} \hspace{1em} A symbol.

\textbf{Description} \hspace{1em} The macro \texttt{define-atomic-modify-macro} has the same syntax as \texttt{cl:define-modify-macro}, and performs a similar operation.

The resulting macro \textit{name} can be used only on one of the specific set of places defined for low level atomic operations as listed in “Low level atomic operations” on page 299. The macro \textit{name} reads the value of the \textit{place}, calls the function \textit{function}, and then writes the result of the function call if the value in \textit{place} has not changed since it was first read. If that value did change, the operation is repeated until it succeeds.

Note that this means:

1. The function \textit{function} may be called more than once for each invocation of the defined macro. Therefore \textit{function} should not have any side effects.

2. \textit{function} must be thread-safe, because it may run concurrently in several threads if the defined macro \textit{name} is used from several threads simultaneously.

3. It is possible in principle for the value to change more than once between reading the \textit{place} and writing the new value. This may end up resetting the value in \textit{place} to its original value, and hence the operation will succeed. This is equivalent to the code being invoked after the last change, unless \textit{function} itself looks at \textit{place}, which may cause inconsistent results.

\textbf{See also} \hspace{1em} \texttt{low-level-atomic-place-p}
**define-top-loop-command**

**Macro**

**Summary**
Defines a top level loop command.

**Package**
`system`

**Signature**
```
define-top-loop-command name-and-options lambda-list form*
name-and-options ::= name
| (name option*)
option ::= (:aliases alias*)
| (:result-type result-type)
```

**Arguments**
- `name`: A keyword naming the command.
- `alias`: A keyword naming an alias for the command.
- `lambda-list`: A destructuring lambda list.
- `result-type`: One of the symbols `values`, `eval` and `nil`.

**Description**
The macro `define-top-loop-command` defines a top level loop command called `name` which takes the parameters specified by `lambda-list`. If `&whole` is used in `lambda-list` then the variable will be bound to a list containing the whole command line, including the command name, but the command name is not included in `lambda-list` otherwise.

If any aliases are specified in `option`, these keywords will also invoke the command.

When the command is used, each form is evaluated in sequence with the variables from `lambda-list` bound to the subsequent forms on the command line.

If `result-type` is `values` (the default), then the values of the last form will be returned to the top level loop.
If \textit{result-type} is \\texttt{eval}, then the value of the last form should be
a form and is evaluated by the top level loop as if it had been
entered at the prompt.

If \textit{result-type} is \texttt{nil}, then the last form should return two val-
ues. If the second value is \texttt{nil} then the first value is treated
as a list of values to returned to the top level loop. If the sec-
ond value is non-nil then the first value should be a form and
is evaluated by the top level loop as if it had been entered at
the prompt.

Notes

For details of pre-defined top level loop commands, enter :? at the Listener prompt.

Example

Given this definition:

\begin{verbatim}
(define-top-loop-command (:lave
    (:result-type eval)) (form)
    (reverse form))
\end{verbatim}

then the command line

\texttt{:lave (1 2 list)}

will evaluate the form \texttt{(list 2 1)}.

Here are definitions for two commands both of which will run \texttt{apropos}:

\begin{verbatim}
(define-top-loop-command (:apropos-eval
    (:result-type eval))
    (&rest args)
    ~(apropos ,@args))
\end{verbatim}

\begin{verbatim}
(define-top-loop-command :apropos-noeval (&rest args)
    (apply 'apropos args))
\end{verbatim}

The first one will evaluate the arguments before calling
\texttt{apropos} whereas the second one will just pass the forms, so

\texttt{:apropos-noeval foo}

will find all the symbols containing the string \texttt{foo}, whereas

\texttt{(setq foo "bar")}
:apropos-eval foo

will find all the symbols containing the string bar.

detect-eol-style

Function

Summary

Detects the end of line style of a file.

Package

system

Signature

detect-eol-style pathname ef-spec buffer length => new-ef-spec

Arguments

pathname Pathname identifying location of buffer.

ef-spec An external format spec.

buffer A buffer whose contents are examined.

length Length (an integer) up to which buffer should be examined.

Values

new-ef-spec A new external format spec created by merging ef-spec with the encoding that was found.

Description

When the encoding in ef-spec has foreign type (unsigned-byte 8), search buffer up to length for the first occurrence of the byte (10). If found, and it is preceded in buffer by (13), merge ef-spec with

(:default :eol-style :crlf)

If found and is not preceded by (13), merge ef-spec with

(:default :eol-style :lf)

Thus a complete external format spec is constructed. Otherwise, return ef-spec.
When the encoding in ef-spec has foreign type \texttt{(unsigned-byte 16)}, search buffer up to length for the first occurrence of the byte sequence \texttt{(13 0 10)}. If found, merge ef-spec with

\texttt{(:default :eol-style :crlf)}

If \texttt{(13 0 10)} is not found, search buffer up to length for \texttt{(10 0)} or \texttt{(0 10)}. If found, merge ef-spec with

\texttt{(:default :eol-style :lf)}

Thus a complete external format spec is constructed. Otherwise, return ef-spec.

See also \texttt{*file-eol-style-detection-algorithm*}

**detect-japanese-encoding-in-file**

\textbf{Function}

\textbf{Summary}\hspace{1cm} Determines which type of Japanese encoding is used in a buffer.

\textbf{Package} \hspace{1cm} system

\textbf{Signature} \hspace{1cm} detect-japanese-encoding-in-file pathname ef-spec buffer length => new-ef-spec

\textbf{Arguments} \hspace{1cm} \begin{itemize}
\item \texttt{pathname} \hspace{1cm} Pathname identifying location of buffer.
\item \texttt{ef-spec} \hspace{1cm} An external format spec.
\item \texttt{buffer} \hspace{1cm} A buffer whose contents are examined.
\item \texttt{length} \hspace{1cm} Length (an integer) up to which buffer should be examined.
\end{itemize}

\textbf{Values} \hspace{1cm} \texttt{new-ef-spec} \hspace{1cm} A new external format spec created by merging ef-spec with the Japanese encoding that was found.
Assume the encoding is one of \texttt{jis}, \texttt{sjis}, \texttt{euc}, \texttt{unicode} and \texttt{ascii}, and try to determine which of these it is, by looking for distinctive byte sequences in \texttt{buffer} up to \texttt{length}. If found, merge \texttt{ef-spec} with that encoding.

See also \*file-encoding-detection-algorithm\*

**Functions**

**detect-unicode-bom**
**detect-utf32-bom**
**detect-utf8-bom**

**Summary**
Looks for the Unicode Byte Order Mark, which if found is assumed to indicate the matching Unicode encoding.

**Package**
\texttt{system}

**Signature**
\texttt{detect-unicode-bom} \texttt{pathname ef-spec buffer length => new-ef-spec}
\texttt{detect-utf32-bom} \texttt{pathname ef-spec buffer length => new-ef-spec}
\texttt{detect-utf8-bom} \texttt{pathname ef-spec buffer length => new-ef-spec}

**Arguments**
- \texttt{pathname} Pathname identifying the location of \texttt{buffer}.
- \texttt{ef-spec} An external format spec.
- \texttt{buffer} A buffer whose contents are examined.
- \texttt{length} Length (an integer) up to which \texttt{buffer} should be examined.

**Values**
- \texttt{new-ef-spec} A new external format spec created by merging \texttt{ef-spec} with the encoding that was found.

**Description**
These functions are called as part of \texttt{open\textsc{'s}} encoding detection routine, and try to detect the encoding if it is not already supplied in the \texttt{external-format} argument.
detect-unicode-bom tries to detect UTF-16 encoding.
detect-utf32-bom tries to detect UTF-32 encoding.
detect-utf8-bom tries to detect UTF-8 encoding.

These functions work by checking whether the file starts with the Unicode character \#xFEFF (BOM) encoded in the relevant encoding, and if it does assumes the file is encoded in this encoding. detect-unicode-bom and detect-utf32-bom also deduce the direction (little-endian or big-endian).

Note that files starting with 0xff 0xfe 0x00 0x00 can match both UTF-16 and UTF-32 little-endian. By default detect-utf32-bom is applied first, because it precedes detect-unicode-bom in *file-encoding-detection-algorithm*. You can change this behavior by altering the order of functions in *file-encoding-detection-algorithm*.

See also *file-encoding-detection-algorithm*

**directory-link-transparency**

Variable

Summary Controls whether directory returns truenames on Unix-like systems.

Package system

Initial value t on non-Windows platforms, nil on Microsoft Windows.

Description In line with the ANSI Common Lisp standard, directory returns truenames by default.

Setting *directory-link-transparency* to nil allows you to get the old behavior of directory, whereby soft links are not resolved in the pathnames returned.
*directory-link-transparency* is the default value of the link-transparency argument to directory.

See also directory

**ensure-loads-after-loads**

*Function*

**Summary**
Ensures all following loads in the program are executed after all prior loads.

**Package**
*system*

**Signature**
ensure-loads-after-loads => nil

**Description**
ensure-loads-after-loads is a synchronization function which ensures order of memory between operations in different threads.

See “Ensuring order of memory between operations in different threads” on page 301 for a full description and example.

**Notes**
You should have a good understanding of multiprocessing issues at the CPU level to write code that actually needs this.

See also
ensure-memory-after-store
ensure-stores-after-memory
ensure-stores-after-stores

**ensure-memory-after-store**

*Function*

**Summary**
Ensures all following stores and loads in the program are executed after all prior stores.

**Package**
*system*
### Signature

| ensure-memory-after-store => nil |

### Description

**ensure-memory-after-store** is a synchronization function which ensures order of memory between operations in different threads.

See “Ensuring order of memory between operations in different threads” on page 301 for a full description and example.

### Notes

You should have a good understanding of multiprocessing issues at the CPU level to write code that actually needs this.

### See also

- ensure-loads-after-loads
- ensure-stores-after-memory
- ensure-stores-after-stores

---

### ensure-stores-after-memory

**Function**

### Summary

Ensures all following stores in the program are executed after all prior stores and loads.

### Package

system

### Signature

| ensure-stores-after-memory => nil |

### Description

**ensure-stores-after-memory** is a synchronization function which ensures order of memory between operations in different threads.

See “Ensuring order of memory between operations in different threads” on page 301 for a full description and example.

### Notes

You should have a good understanding of multiprocessing issues at the CPU level to write code that actually needs this.
See also

- ensure-loads-after-loads
- ensure-memory-after-store
- ensure-stores-after-stores

**ensure-stores-after-stores**

*Function*

**Summary**
Ensures all following stores in the program are executed after all prior stores.

**Package**
`system`

**Signature**
`ensure-stores-after-stores => nil`

**Description**
`ensure-stores-after-stores` is a synchronization function which ensures order of memory between operations in different threads.

See “Ensuring order of memory between operations in different threads” on page 301 for a full description and example.

**Notes**
You should have a good understanding of multiprocessing issues at the CPU level to write code that actually needs this.

See also

- ensure-loads-after-loads
- ensure-memory-after-store
- ensure-stores-after-memory

**`extended-spaces`**

*Variable*

**Summary**
Extends the notion of space to include more than just the space character.

**Package**
`system`

**Initial value**
`nil`
Description

When this variable is true, the concept of “space” is extended from just `\Space` to include other appropriate characters. The default is `nil`, for ANS compliance, but we recommend that you set it to `t`.

This variable controls how the format directives `~:C` and `~:@C` output graphic characters which have an empty glyph. When this variable is `t`, all such characters are output using the name:

- `(format nil "~:C" #\No-break-space)` -> "No-Break-Space"
- `(format nil "~:C" (code-char #x3000))` -> "Ideographic-Space"

When false, only one such character is output using the name:

- `(format nil "~:C" \Space)` -> "Space"
- `(format nil "~:C" \No-break-space)` -> " "
- `(format nil "~:@C" (code-char #x3000))` -> "  "

It also affects `whitespace-char-p`.

See also `whitespace-char-p`

*file-encoding-detection-algorithm* *Variable*

Summary

List of functions to call to work out an encoding.

Package `system`

Initial value `(find-filename-pattern-encoding-match find-encoding-option detect-utf32-bom detect-unicode-bom detect-utf8-bom specific-valid-file-encoding locale-file-encoding)`
Functions on this list take four arguments—the pathname of the file; an external format spec; a vector of element-type
(unsigned-byte 8) which contains the first bytes of the file; and a non-negative integer which is the maximum extent of buffer to be searched. This length argument is 0 in the case that the file does not exist, or the direction is :output. They return an external format spec, which normally is either ef-spec unmodified, or the result of merging ef-spec with another external format spec via merge-ef-specs.

See the entry for guess-external-format for details of how *file-encoding-detection-algorithm* is used.

For files starting with 0xff 0xfe 0x00 0x00, both detect-utf32-bom and detect-unicode-bom may match it. detect-utf32-bom is called first so by default the encoding will be detected as (:utf-32 :little-endian t). You can change this behavior by setting *file-encoding-detection-algorithm* to a re-ordered list.

If you want open and so on, when opening a file for input, to inspect the attribute line and then fall back to a default if no attribute line is found, then set the variable to this value:

(find-encoding-option locale-file-encoding)

There are further examples in “Guessing the external format” on page 448.

See also find-filename-pattern-encoding-match
find-encoding-option
detect-unicode-bom
detect-japanese-encoding-in-file
guess-external-format
locale-file-encoding
**file-encoding-resolution-error**  

**Condition**

**Summary**  
An error type to signal when an external file format cannot be deduced.

**Package**  
`system`

**Superclasses**  
`error`

**Initargs**  
`:ef-spec`  
An external format specification.

**Description**  
An error type signaled when `open`, `load` or `compile-file` fail to detect an external format to use.

The `ef-spec` slot contains the incomplete external format specification argument constructed by `guess-external-format`.

**See also**  
`guess-external-format`

**file-eol-style-detection-algorithm**  

**Variable**

**Summary**  
List of functions for determining the end of line style of a file.

**Package**  
`system`

**Description**  
Functions on this list satisfy the same specifications as for those in `*file-encoding-detection-algorithm*`. However, they will only be passed an external format spec with the name already determined.

**Initial value**  
`(detect-eol-style default-eol-style)`

**See also**  
`detect-eol-style`  
`default-eol-style`  
`guess-external-format`
*filename-pattern-encoding-matches*  

Summary  
An association of filename patterns to external format specs.

Package  
*system*

Initial value  
`((TAGS . (:latin-1 :eol-style :lf)))`

Description  
An alist of filename patterns to external format specs.

See also  
*file-encoding-detection-algorithm*

find-encoding-option  

Summary  
Examines a buffer for an encoding option.

Package  
*system*

Signature  

```
find-encoding-option pathname ef-spec buffer length => result
```

Arguments  
```
pathname  Pathname identifying location of buffer.

ef-spec  An external format spec.

buffer  A buffer whose contents are examined.

length  Length (an integer) up to which buffer should be examined.
```

Values  
```
result  The result of reading the value returned from the encoding or external-format option as a Lisp expression in the keyword package.
```

Description  
Looks in the file options (EMACS-style -*- line) for an option called encoding or external-format, with value value.
If `encoding` or `external-format` is found, it reads `value` as a Lisp expression in the `keyword` package. If `coding` is found, it attempts to translate `value` from a GNU Emacs coding system name to a LispWorks external-format name.

It then merges the `ef-spec` with the external format spec derived from `value`, and returns the result as `result`. Thus it does not override a supplied `ef-spec`.

See also *file-encoding-detection-algorithm*

**find-filename-pattern-encoding-match**

**Function**

**Summary**
Finds the encoding of a file based on the filename.

**Package**
`system`

**Signature**
`find-filename-pattern-encoding-match pathname ef-spec buffer length => new-ef-spec`

**Arguments**
- `pathname`: Pathname identifying location of `buffer`.
- `ef-spec`: An external format spec.
- `buffer`: A buffer whose contents are examined.
- `length`: Length (an integer) up to which `buffer` should be examined.

**Values**
- `new-ef-spec`: An external format spec.

**Description**
Compares `pathname` (using `pathname-match-p`) with elements of `*filename-pattern-encoding-matches*`.

If a match is found, merges `ef-spec` with the corresponding external format spec and returns the result as `new-ef-spec`. Thus it does not override a supplied `ef-spec`.
See also *file-encoding-detection-algorithm*  
*filename-pattern-encoding-matches*

**gen-num-segments-fragmentation-state**  
*Function*

**Summary**  
Shows the fragmentation state in a generation in 64-bit Lisp-Works.

**Package**  
system

**Signature**  
gen-num-segments-fragmentation-state gen-num &optional statics-too => fragmentation-state

**Arguments**  
gen-num A number.  
statics-too A generalized boolean?

**Values**  
fragmentation-state  
A list in which each element is a list of length 3.

**Description**  
The function `gen-num-segments-fragmentation-state` shows the fragmentation state in a generation in 64-bit Lisp-Works.

`gen-num-segments-fragmentation-state` returns a list, where each element is a sub-list showing the fragmentation state in a segment. The sub-list is of the form

```
(allocation-type allocated free)
```

where `allocation-type` is the allocation type of the segment,  
`allocated` is the amount of allocated data in the segment, and  
`free` is the total size of free areas in the segment that cannot be easily used.

The ratio `free/allocated` is the ratio that is compared to the fragmentation threshold to decide whether to copy a segment
when doing a marking GC with copying (see \texttt{set-blocking-gen-num} and \texttt{marking-gc}).

Allocation types \texttt{:cons-static}, \texttt{:non-pointer-static}, \texttt{:mixed-static}, \texttt{:other-big} and \texttt{:non-pointer-big} are included in the result only if \texttt{statics-too} is non-nil. The default value of \texttt{statics-too} is \texttt{nil}.

Notes

1. The implementation of \texttt{set-blocking-gen-num} is intended to solve any fragmentation issues automatically.

2. \texttt{gen-num-segments-fragmentation-state} is implemented only in 64-bit LispWorks. It does nothing in the Mobile GC and its return value is not meaningful. It is not relevant to the Memory Management API in 32-bit implementations, where \texttt{check-fragmentation} is available instead.

See also

\texttt{check-fragmentation}

\texttt{marking-gc}

\texttt{set-blocking-gen-num}

“Guidance for control of the memory management system” on page 130

\begin{verbatim}
function
generation-number

Summary
\texttt{Returns the current generation number for an object.}

Package
\texttt{system}

Signature
\texttt{generation-number object => integer}

Arguments
\texttt{object} \hspace{1cm} A Lisp object.

Values
\texttt{integer} \hspace{1cm} An integer.
\end{verbatim}
The function `generation-number` returns the generation number in which the Lisp object `object` currently is. See the discussion in Chapter 11, “Memory Management”.

If `object` is an immediate object then `generation-number` returns -1. Immediates are objects which are not allocated, including fixnums, characters and short floats, and single floats in 64-bit LispWorks.

See also “Guidance for control of the memory management system” on page 130

`gesture-spec-accelerator-bit`  
`gesture-spec-caps-lock-bit`  
`gesture-spec-control-bit`  
`gesture-spec-hyper-bit`  
`gesture-spec-meta-bit`  
`gesture-spec-shift-bit`  
`gesture-spec-super-bit`  

**Constants**

**Summary**

Used in the representation of keystrokes with the various modifier keys.

**Package**  
`system`

**Description**

These constants are used to represent the accelerator and modifier keys in a Gesture Spec object, as follows:

`gesture-spec-accelerator-bit`  
Accelerator key.

`gesture-spec-caps-lock-bit`  
Caps Lock modifier key.

`gesture-spec-control-bit`  
Control modifier key.
gesture-spec-hyper-bit

Hyper modifier key.

gesture-spec-meta-bit

Meta modifier key.

gesture-spec-shift-bit

Shift modifier key.

gesture-spec-super-bit

Super modifier key.


Notes

1. You may not construct a Gesture Spec with a both-case-p character represented in the data and with modifiers equal to gesture-spec-shift-bit. See make-gesture-spec for details and examples.

2. The gesture-spec-caps-lock-bit is used to represent the state of Caps Lock, in situations where the bits are used to represent the keyboard state. It is not used in Gesture Specs that are generated by the system.

3. The gesture-spec-hyper-bit is used to represent the Command key.

4. The gesture-spec-accelerator-bit is a "virtual" bit. It corresponds to different keys on different GUI systems, currently these are Command on Cocoa, Control on GTK+ and Control on Windows.

See also

c coerce-to-gesture-spec
gesture-spec-modifiers
make-gesture-spec
**gesture-spec-data**

*Function*

Summary: Returns the key in a Gesture Spec object.

Package: `system`

Signature: `gesture-spec-data gspec => data`

Arguments: `gspec` A Gesture Spec object

Values: `data` A non-negative integer or a keyword.

Description: The function `gesture-spec-data` returns an integer or keyword representing the key in the Gesture Spec object `gspec`.

When `data` is an integer, it is a non-negative integer less than `cl:char-code-limit`, and `gspec` represents a keystroke with the key indicated by the character which is the value of `(code-char data)`.

`data` can also be a keyword such as `:f6`, when `gspec` represents a keystroke with `F6` pressed.

See also: `gesture-spec-modifiers`  
`make-gesture-spec`

**gesture-spec-modifiers**

*Function*

Summary: Returns the modifiers in a Gesture Spec object.

Package: `system`

Signature: `gesture-spec-modifiers gspec => mods`

Arguments: `gspec` A Gesture Spec object

Values: `mods` An integer.
Description

The function `gesture-spec-modifiers` returns an integer representing the modifiers in the Gesture Spec object `gspec`.

The value `mods` contains some (or none) of the constants `gesture-spec-accelerator-bit`, `gesture-spec-control-bit`, `gesture-spec-meta-bit`, `gesture-spec-hyper-bit`, `gesture-spec-shift-bit` and `gesture-spec-super-bit`, combined as if by `logior`.

See also

`gesture-spec-accelerator-bit`
`gesture-spec-control-bit`
`gesture-spec-data`
`gesture-spec-meta-bit`
`gesture-spec-hyper-bit`
`gesture-spec-shift-bit`
`gesture-spec-super-bit`
`make-gesture-spec`

---

**gesture-spec-p**

**Function**

**Summary**

The predicate for Gesture Spec objects.

**Package**

`system`

**Signature**

`gesture-spec-p object => result`

**Arguments**

`object` A Lisp object

**Values**

`result` A boolean.

**Description**

The function `gesture-spec-p` is the predicate for whether the object `object` is a Gesture Spec object.

See also

`coerce-to-gesture-spec`
`make-gesture-spec`
gesture-spec-to-character  

**Function**

**Summary**
Returns the character corresponding to a Gesture Spec object.

**Package**
`system`

**Signature**
`gesture-spec-to-character gspec &key errorp => char`

**Arguments**
gspec  
A Gesture Spec object

**Values**
char  
A Lisp character.

errorp  
A generalized boolean.

**Description**
The function `gesture-spec-to-character` returns the Lisp `character` object corresponding to the Gesture Spec object `gspec`.

A Gesture Spec with modifiers or data which is not an integer cannot be converted to a character. When supplied such a Gesture Spec, `gesture-spec-to-character` either signals an error (if `errorp` is true), or returns `nil` (if `errorp` is `nil`). The default value of `errorp` is `t`.

`gesture-spec-accelerator-bit` is ignored.

**Compatibility note**
In LispWorks 6.1 and earlier versions, `gesture-spec-to-character` allows modifiers and does not error when the data is not an integer. LispWorks 7.0 and later versions do not support character bits, therefore if `gspec` contains non-zero modifier bits, `gesture-spec-to-character` signals an error.

**See also**
`coerce-to-gesture-spec`
`make-gesture-spec`
**get-file-stat**  

**Summary**  
Provides read access to the C stat structure which describes files.  

**Note:** not applicable on Microsoft Windows.

**Package**  
`system`

**Signature**  
`get-file-stat filename-or-fd => file-stat(errno)`

**Arguments**  
`filename-or-fd`  
A string denoting a file, or a file descriptor.

**Values**  
`file-stat`  
On success, an object representing the stat values. On failure, `nil` is returned together with a second value.

`errno`  
Indicates the `errno` value returned by the system call. This second value is returned only in the case of failure.

**Description**  
`file-stat` is an object representing the stat values, as would be returned by the system call `stat` (for a filename) or the system call `fstat` (for an fd).

The values in `file-stat` are the raw data, and it is the responsibility of the user to interpret them when needed. See the UNIX manual entry for `stat` for details.

The values can be read from `file-stat` by these readers:

`sys:file-stat_inode`  
The inode of the file.

`sys:file-stat-device`  
The id of the device where the file is.

`sys:file-stat-owner-id`  
The user id of the owner of the file.
**sys:file-stat-group-id**
The group id of the file's group.

**sys:file-stat-size**
The size of the file in bytes.

**sys:file-stat-blocks**
The number of 512-bytes blocks used by the file.

**sys:file-stat-mode**
The protection value of the file.

**sys:file-stat-last-access**
The time of the last access to the file in seconds from 1 January 1970.

**sys:file-stat-last-change**
The time of the last change in the data of the file in seconds from 1 January 1970.

**sys:file-stat-last-modify**
The time of the last modification of the file status in seconds from 1 January 1970.

**sys:file-stat-links**
The number of hard links to the file.

**sys:file-stat-device-type**
The device type (sometimes called Rdev).

---

**get-folder-path**

**Function**

**Summary**
Gets the path of a special folder.

**Package**
`system`
Signature  
\[
\text{get-folder-path}\; \text{what} \; \&\text{key} \; \text{create} \; \Rightarrow \; \text{result}
\]

Arguments  
\[
\begin{align*}
\text{what} & \quad \text{A keyword.} \\
\text{create} & \quad \text{A boolean.}
\end{align*}
\]

Values  
\[
\begin{align*}
\text{result} & \quad \text{A directory pathname naming the path, or nil.}
\end{align*}
\]

Description  
The function \text{get-folder-path} obtains the current value for various special folders often used by applications. It is useful because these paths may differ between versions of the operating system. \text{get-folder-path} is implemented all platforms, using system APIs on Microsoft Windows, Mac OS X and Android.

On platforms other than Windows, Mac OS X and Android it is a dummy function, which makes a path to a directory inside the user’s home directory that looks like \text{<homedir>/get-folder-path/<symbol-name-down-cased>}. This allows testing code that uses \text{get-folder-path} to work in the sense that files can be written and read from these directories.

\text{what} indicates the purpose of the special folder. For instance, \text{:common-appdata} means the folder containing application data for all users.

The following values of \text{what} are recognized on Microsoft Windows and Mac OS X:

\text{:appdata, :documents, :my-documents, :common-appdata, :common-documents} and \text{:local-appdata, :documents} is an alias for \text{:my-documents}.

The following values are recognized on Microsoft Windows only: \text{:program-files, :programs} and \text{:common-programs}.

The following values are recognized on Mac OS X only:
The SYSTEM Package


On Mac OS X, :appdata is an alias for :my-appsupport, :common-appdata is an alias for :common-appsupport, and :local-appdata is an alias for :common-appsupport.

If the folder does not exist and create is true, the folder is created. If the folder does not exist and create is false, result is nil. The default value of create is false.

The following values of what are recognized on Android:

:appdata, :local-appdata

Both of these return the same directory. It is the directory which is returned by the getFilesDir on the application context. Note that this is a private directory, not visible to other applications.

:my-documents, :documents

On Android 4.4 and later this returns the "documents" directory in the "public external" directory (the result of calling "android.os.Environment.getExternalStoragePublicDirectory" with the value of android.os.Environment.DIRECTORY_DOCUMENTS). In previous versions it uses the "downloads" directory, because there does not seem to be another useful place for it.

Note: This is used as the home directory on Android, that is what cl:user-homedir-pathname returns.


Return the matching directory in the "public external" directory. This is the result of calling "android.os.Environment.getExternalStoragePublicDirectory" with the value of android.os.Environment.DIRECTORY_<symbol-name>, where symbol-name is the symbol name of the key-
word), for example `android.os.Environment.DIRECTORY_RINGTONES`

:common-appdata

Returns the external storage directory of the application if it is accessible, otherwise returns `nil`. The external storage directory is the result of calling `getExternalFilesDir` on the application context with `null`.

Note that the application will need permission to access the external storage, by having uses-permission `android.permission.WRITE_EXTERNAL_STORAGE` or `android.permission.READ_EXTERNAL_STORAGE` in the `AndroidManifest.xml` file.

**Compatibility notes**

1. In LispWorks 6.1 and earlier versions, `get-folder-path` is implemented only on Windows and Mac OS X.

2. In LispWorks 5.0 and previous versions, `get-folder-path` returns a string.

**Example**

This form constructs a pathname to a file `foo.lisp` in the user's documents directory:

```lisp
(make-pathname
 :name "foo"
 :type "lisp"
 :defaults
 (sys:get-folder-path :my-documents))
```

**See also**

`get-user-profile-directory`

---

**get-maximum-allocated-in-generation-2-after-gc**

**Function**

**Summary**

Mobile GC only: Returns the maximum number of allocated bytes in generation 2 immediately after a GC.

**Package**

`system`
Signature  
\texttt{get-maximum-allocated-in-generation-2-after-gc}  
\&optional \texttt{reset-p} \to \texttt{other-size-in-bytes, cons-size-in-byte, gen-2-gc-count}

Arguments  
\texttt{reset-p} \hspace{1cm} \text{Boolean.}

Values  
\texttt{other-size-in-bytes, cons-size-in-byte, gen-2-gc-count}  
\hspace{1cm} \text{Integers.}

Description  
The function \texttt{get-maximum-allocated-in-generation-2-after-gc} returns the maximum bytes used by live objects in generation 2 immediately after any of the preceding GCs of generation 2 since the previous "reset" (a call to \texttt{get-maximum-allocated-in-generation-2-after-gc} with \texttt{reset-p} non-nil). It also returns the number of GCs of generation 2 since the previous reset.

\texttt{other-size-in-bytes} is the maximum size of live objects in Other segments (that is not Cons, Large or Static) immediately after any of these GCs, and \texttt{cons-size-in-byte} is the maximum size of live conses.

\texttt{gen-2-gc-count} if the number of GCs of generation 2 that have occurred since the last call with \texttt{reset-p} non-nil.

The values of \texttt{other-size-in-bytes} and \texttt{cons-size-in-byte} match the values that would have been reported for generation 2 by \texttt{room}, if it had been called immediately after a GC of generation 2.

\texttt{reset-p} defaults to \texttt{nil}. When it is non-nil, the maximums and count are reset to 0.

Notes  
The purpose of \texttt{get-maximum-allocated-in-generation-2-after-gc} is to give useful information for controlling generation 2, for example to decide what values to use in \texttt{set-expected-allocation-in-generation-2-after-gc}. This function is also useful for just counting the number of GCs of generation 2.
get-user-profile-directory

Function

Summary

Gets the root of the user’s profile on a Windows system.

Package

system

Signature

get-user-profile-directory => result

Values

result

A directory pathname naming the path, or nil.

Description

The function get-user-profile-directory obtains the path to the current user’s profile folder on a Windows system. get-user-profile-directory is implemented only on Microsoft Windows.

result names the root of the profile directory.

Note that the default path for each user’s profile may differ between versions of the operating system.

Compatibility notes

In LispWorks 5.0 and previous versions, get-user-profile-directory returns a string.

Example

On Windows 10, Windows 8 and Windows 7:

(sys:get-user-profile-directory)
=>
#P"C:/Users/dubya/"

On Windows XP (now unsupported):

(sys:get-user-profile-directory)
=>
#P"C:/Documents and Settings/dubya/"

See also

get-folder-path
**globally-accessible**

*Macro*

**Summary**
A wrapper `setf` place that ensures earlier stores are visible to other threads before storing into the inner place.

**Package**
`system`

**Signature**
`globally-accessible place => value`

**Arguments**
`place`  
A generalized reference form as described in section 5.1.1 Overview of Places and Generalized Reference of the Common Lisp Hyperspec.

**Values**
`value`  
Any Lisp object.

**Description**
The macro `globally-accessible` expands to `place`. The effect of using `(globally-accessible place)` is the same as `place`, except when used inside `setf` or a related macro such as `push` or `incf` where it also ensures all stores are visible to other threads before modifying `place`. This includes all the stores that were made into the new value and, for a modifying macro or complex accessor, any stores that are done by the expansion.

See “Ensuring stores are visible to other threads” on page 271 for a full discussion when `globally-accessible` is needed.

When used with accessors that take a place as argument (`getf`, `mask-field`, `ldb` or `cdr-assoc`), `globally-accessible` needs to be used around innermost place, rather than the accessor, for example:

```lisp
(setf (getf (sys:globally-accessible *a-global-symbol*)
    key)
    value)
```

rather than:
(setf ; WRONG
     (sys:globally-accessible
      (getf *a-global-symbol* key))
     value)

globally-accessible tries to avoid ensuring all stores when it is possible to avoid it, for example when used inside pushnew if the value is already in the list.

Notes

You do not need to use globally-accessible when any of the following apply:

• place can be accessed only by the same thread that stores into it (so it is not globally accessible). This is the common situation for stores.

• Access to place (both read and writes) is synchronized between threads, normally by a lock but maybe by some other synchronization mechanism. This is the preferred way to access globally accessible cells.

• The store is done by one of: (setf gethash), vector-push, vector-push-extend, (setf symbol-function), (setf macro-function) and the hash-table or vector containing the globally accessible cell was not created as single-threaded.

In other cases (globally accessible cells which are read without synchronization), you probably need globally-accessible. See “Ensuring stores are visible to other threads” on page 271 for exact details.

See also

“Ensuring stores are visible to other threads” on page 271

guess-external-format  

Function

Summary  
Tries to work out the external format

Package  
system
Signature  
\texttt{guess-external-format} \textit{pathname ef-spec buffer length} => \textit{ef-spec}

Arguments  
\texttt{pathname} Pathname identifying location of buffer.
\texttt{ef-spec} An external format spec.
\texttt{buffer} A buffer whose contents are examined.
\texttt{length} Length (an integer) up to which buffer should be examined.

Values  
\texttt{ef-spec} An external format spec.

Description  
If \texttt{ef-spec} is complete, then it is returned. Otherwise \texttt{guess-external-format} calls, in turn, functions on the list \texttt{*file-encoding-detection-algorithm*}. If a complete external format spec is returned it is used, otherwise the return value is passed to the next function. If the name of the external format spec returned by the last function on this list is \texttt{:default}, an error of type \texttt{file-encoding-resolution-error} is signaled. Otherwise \texttt{guess-external-format} proceeds to guess the \texttt{eol-style}.

To guess the \texttt{eol-style}, functions on the list \texttt{*file-eol-style-detection-algorithm*} are called in turn. If a complete external format spec is returned it is used, otherwise the return value is passed to the next function. If the external format spec returned by the last function on this list does not contain \texttt{:eol-style}, an error of type \texttt{file-encoding-resolution-error} is signaled.

See also  
\texttt{*file-encoding-detection-algorithm*}
\texttt{*file-eol-style-detection-algorithm*}
\texttt{file-encoding-resolution-error}

\textbf{immediatep}  
Function

Summary  
The predicate for immediate objects.
### immediatep

**Package**  
`system`

**Signature**  
`immediatep object => result`

**Arguments**  
`object`  
A Lisp object.

**Values**  
`result`  
A boolean.

**Description**  
The function `immediatep` returns `t` for "immediate" objects, that is objects that do not actually use heap memory.

### in-static-area

**Macro**

**Summary**  
Allocates the objects produced by the specified forms to the static area.  
This macro is deprecated.

**Package**  
`system`

**Signature**  
`in-static-area &rest body => result`

**Arguments**  
`body`  
The forms for which you want the garbage collector to allocate space in the static area.

**Values**  
`result`  
The result of executing `body`.

**Description**  
Allocates the objects produced by the specified forms to the static area. Objects in the static area are not moved, though they are garbage collected when there is no longer a pointer to the object.

**Notes**  
`in-static-area` is deprecated. Use `make-array` with `:allocation :static` where possible instead.
In 64-bit LispWorks and the Mobile GC, \texttt{in-static-area} does not affect the allocation conses. There is no interface to make static conses in 64-bit LispWorks or the Mobile GC.

Example

\begin{verbatim}
(system:in-static-area (make-string 10))
\end{verbatim}

See also

\begin{itemize}
  \item \texttt{enlarge-static}
  \item \texttt{make-array}
  \item \texttt{staticp}
\end{itemize}

\textbf{int32}

\textit{Type}

\textbf{Summary}

A type used to generate optimal 32-bit arithmetic code.

\textbf{Package}

\texttt{system}

\textbf{Signature}

\texttt{int32}

\textbf{Description}

The type \texttt{int32} is used to generate optimal 32-bit arithmetic code.

Objects of type \texttt{int32} are generated and can be manipulated using the functions in the INT32 API but the compiler can optimize such source code by eliminating the intermediate \texttt{int32} objects to produce efficient raw 32-bit code.

See the section “Fast 32-bit arithmetic” on page 472 for more information.

See also

\begin{itemize}
  \item \texttt{int32*}
  \item \texttt{+int32-0+}
  \item \texttt{+int32-1+}
  \item \texttt{int32-1+}
  \item \texttt{int32/=}
  \item \texttt{int32<}
  \item \texttt{int32-aref}
  \item \texttt{int32-logand}
\end{itemize}
The arithmetic operators for int32 objects.

**Summary**
The function int32* is the multiply operator for int32 objects.
The function int32+ is the add operator for int32 objects.
The function int32- is the subtract operator for int32 objects.

**Package**
*system*

**Signatures**
- int32* x y => int32
- int32+ x y => int32
- int32- x y => int32
- int32/ x y => int32

**Arguments**
- \(x\)  
  An int32 object or an integer of type (signed-byte 32).
- \(y\)  
  An int32 object or an integer of type (signed-byte 32).

**Values**
- int32  
  An int32 object.
The function `int32/` is the divide operator for `int32` objects. See the section “Fast 32-bit arithmetic” on page 472 for more information about the INT32 API.

See also `int32`

**+int32-0+**

*Symbol Macro*

**Summary** Shorthand for `(sys:integer-to-int32 0)`.

**Package** system

**Description** The symbol macro `+int32-0+` expands to `(sys:integer-to-int32 0)`.

See also `integer-to-int32`

**+int32-1+**

*Symbol Macro*

**Summary** Shorthand for `(sys:integer-to-int32 1)`.

**Package** system

**Description** The symbol macro `+int32-1+` expands to `(sys:integer-to-int32 1)`.

See also `integer-to-int32`

**int32-1+**

**int32-1-**

*Functions*

**Summary** The operators for `int32` objects corresponding to the functions 1+ and 1-.
Package system

Signatures

\[
\begin{align*}
\texttt{int32-1+} & \ x \ => \ \texttt{int32} \\
\texttt{int32-1-} & \ x \ => \ \texttt{int32}
\end{align*}
\]

Arguments

\( x \quad \text{An int32 object or an integer of type (signed-byte 32).} \)

Values

\texttt{int32} \quad \text{An int32 object.}

Description

The functions \texttt{int32-1+} and \texttt{int32-1-} are the operators for \texttt{int32} objects corresponding to the functions \(1+\) and \(1-\).

See the section “Fast 32-bit arithmetic” on page 472 for more information about the INT32 API.

See also \texttt{int32}

\[
\begin{align*}
\texttt{int32/=} & \\
\texttt{int32<} & \\
\texttt{int32<=} & \\
\texttt{int32=} & \\
\texttt{int32>} & \\
\texttt{int32>=}
\end{align*}
\]

Functions

Summary

The comparison operators for \texttt{int32} objects.

Package system

Signatures

\[
\begin{align*}
\texttt{int32/=} & \ x \ y \ => \ \texttt{result} \\
\texttt{int32<} & \ x \ y \ => \ \texttt{result} \\
\texttt{int32<=} & \ x \ y \ => \ \texttt{result} \\
\texttt{int32=} & \ x \ y \ => \ \texttt{result} \\
\texttt{int32>} & \ x \ y \ => \ \texttt{result}
\end{align*}
\]
The SYSTEM Package

```
int32> x y => result
int32>= x y => result
```

Arguments

\( x \) An int32 object or an integer of type (signed-byte 32).

\( y \) An int32 object or an integer of type (signed-byte 32).

Values

\( \text{result} \) A boolean.

Description

The function `int32/=` is the not equal comparison for int32 objects.

The function `int32<` is the less than comparison for int32 objects.

The function `int32<=` is the less than or equal comparison for int32 objects.

The function `int32=` is the equal comparison for int32 objects.

The function `int32>` is the greater than comparison for int32 objects.

The function `int32>=` is the greater than or equal comparison for int32 objects.

See the section “Fast 32-bit arithmetic” on page 472 for more information about the INT32 API.

See also

int32

```
int32<<
int32>>
```

Functions

Summary

The shift operators for int32 objects.
### Package

**system**

### Signatures

- `int32<< x y => result`
- `int32>> x y => result`

### Arguments

- **x**
  - An `int32` object or an integer of type 
    `(signed-byte 32)`.
- **y**
  - An `int32` object or an integer of type 
    `(signed-byte 32)`.

### Values

- **result**
  - An `int32` object.

### Description

The function `int32<<` is a shift left operator for `int32` objects.

The function `int32>>` is a shift right operator for `int32` objects.

See the section “Fast 32-bit arithmetic” on page 472 for more information about the INT32 API.

### See also

- `int32`

---

### int32-aref

#### Function

**Summary**

The accessor for a `simple-int32-vector`.

**Package**

**system**

**Signature**

- `int32-aref vector index => int32`
- `(setf int32-aref) x vector index => int32`

**Arguments**

- **vector**
  - An `simple-int32-vector`.
- **index**
  - A non-negative fixnum.
The function `int32-aref` is the accessor for a `simple-int32-vector`. The reader returns an `int32` object for the value at index `index` in `vector`. The writer sets the value at index `index` in `vector` to the `int32` object or integer `x` supplied.

See the section “Fast 32-bit arithmetic” on page 472 for more information about the INT32 API.

See also

- `int32`
- `simple-int32-vector`

### Functions

- `int32-logand`
- `int32-logandc1`
- `int32-logandc2`
- `int32-logbitp`
- `int32-logeqv`
- `int32-logior`
- `int32-lognand`
- `int32-lognor`
- `int32-lognot`
- `int32-logorc1`
- `int32-logorc2`
- `int32-logtest`
- `int32-logxor`

**Summary**

The bitwise logical operators for `int32` objects.

**Package**

`system`
Signatures

\[
\begin{align*}
\text{int32-logand} & \ x \ y \ => \ \text{int32} \\
\text{int32-logandc1} & \ x \ y \ => \ \text{int32} \\
\text{int32-logandc2} & \ x \ y \ => \ \text{int32} \\
\text{int32-logbitp} & \ \text{index} \ x \ => \ \text{result} \\
\text{int32-logeqv} & \ x \ y \ => \ \text{int32} \\
\text{int32-logior} & \ x \ y \ => \ \text{int32} \\
\text{int32-lognand} & \ x \ y \ => \ \text{int32} \\
\text{int32-lognor} & \ x \ y \ => \ \text{int32} \\
\text{int32-lognot} & \ x \ => \ \text{int32} \\
\text{int32-logorc1} & \ x \ y \ => \ \text{int32} \\
\text{int32-logorc2} & \ x \ y \ => \ \text{int32} \\
\text{int32-logtest} & \ x \ y \ => \ \text{result} \\
\text{int32-logxor} & \ x \ y \ => \ \text{int32}
\end{align*}
\]

Arguments

\[\begin{align*}
x & \quad \text{An int32 object or an integer of type (signed-byte 32)}. \\
y & \quad \text{An int32 object or an integer of type (signed-byte 32)}. \\
\text{index} & \quad \text{An int32 object or an integer of type (signed-byte 32)}.\end{align*}\]

Values

\[\begin{align*}
\text{int32} & \quad \text{An int32 object}. \\
\text{result} & \quad \text{An boolean}.\end{align*}\]

Description

The function \texttt{int32-logand} is the bitwise logical ‘and’ operator for \texttt{int32} objects.

The function \texttt{int32-logandc1} is the bitwise logical operator for \texttt{int32} objects which ‘ands’ the complement of \(x\) with \(y\).

The function \texttt{int32-logandc2} is the bitwise logical operator for \texttt{int32} objects which ‘ands’ \(x\) with the complement of \(y\).
The function \texttt{int32-logbitp} is the test for \texttt{int32} objects which returns \texttt{t} if if the bit at index \texttt{index} in \texttt{x} is 1, and \texttt{nil} if it is 0.

The function \texttt{int32-logeqv} is the bitwise logical operator for \texttt{int32} objects which returns the complement of the ‘exclusive or’ of \texttt{x} and \texttt{y}.

The function \texttt{int32-logior} is the bitwise logical ‘inclusive or’ operator for \texttt{int32} objects.

The function \texttt{int32-lognand} is the bitwise logical operator for \texttt{int32} objects which returns the complement of the ‘and’ of \texttt{x} and \texttt{y}.

The function \texttt{int32-lognor} is the bitwise logical operator for \texttt{int32} objects which returns the complement of the ‘inclusive or’ of \texttt{x} and \texttt{y}.

The function \texttt{int32-lognot} is the bitwise logical operator for \texttt{int32} objects which returns the complement of its argument \texttt{x}.

The function \texttt{int32-logorc1} is the bitwise logical operator for \texttt{int32} objects which ‘inclusive ors’ the complement of \texttt{x} with \texttt{y}.

The function \texttt{int32-logorc2} is the bitwise logical operator for \texttt{int32} objects which ‘inclusive ors’ \texttt{x} with the complement of \texttt{y}.

The function \texttt{int32-logtest} is the bitwise test for \texttt{int32} objects which returns \texttt{t} if any of the bits designated by 1 in \texttt{x} is 1 in \texttt{y}, and returns \texttt{nil} otherwise.

The function \texttt{int32-logxor} is the bitwise logical ‘exclusive or’ operator for \texttt{int32} objects.

See the section “Fast 32-bit arithmetic” on page 472 for more information about the INT32 API.

See also \texttt{int32}
### Functions

**int32-minusp**
**int32-plusp**
**int32-zerop**

**Summary**
The `minusp`, `plusp` and `zerop` tests for an `int32` object.

**Package**
`system`

**Signatures**
- `int32-minusp x => result`
- `int32-plusp x => result`
- `int32-zerop x => result`

**Arguments**
- `x` An `int32` object or an integer of type `(signed-byte 32)`.

**Values**
- `result` A boolean.

**Description**
The function `int32-minusp` tests whether its argument `x` is `int32<` than the value of `+int32-0+`.

The function `int32-plusp` tests whether its argument `x` is `int32>` than the value of `+int32-0+`.

The function `int32-zerop` tests whether its argument `x` is `int32=` to the value of `+int32-0+`.

See the section "Fast 32-bit arithmetic" on page 472 for more information about the INT32 API.

**See also**
- `int32`

### Function

**int32-to-int64**

**Summary**
Converts from `int32` to `int64`.

**Package**
`system`
Signature   \texttt{int32-to-int64} $x \Rightarrow y$

Arguments    $x$ An \texttt{int32} object.

Values       $y$ An \texttt{int64} object.

Description  The function \texttt{int32-to-int64} converts the \texttt{int32} object $x$ to the corresponding \texttt{int64} object $y$.

See also    \texttt{int32}  \\
\texttt{int64}  \\
\texttt{int64-to-int32}

\textbf{int32-to-integer} \hspace{1cm} \textit{Function}

Summary      The destructor converting an \texttt{int32} object to an integer.

Package      \texttt{system}

Signature    \texttt{int32-to-integer} \texttt{int32} \Rightarrow \texttt{integer}

Arguments    \texttt{int32} An \texttt{int32} object or an integer of type \texttt{(signed-byte 32)}.

Values       \texttt{integer} An integer of type \texttt{(signed-byte 32)}.

Description  The function \texttt{int32-to-integer} returns an integer \texttt{integer} of type \texttt{(signed-byte 32)} corresponding to the \texttt{int32} object \texttt{int32}. The argument \texttt{int32} can also be an integer of type \texttt{(signed-byte 32)}, in which case it is simply returned.

An error is signaled if \texttt{int32} is not of type \texttt{int32} or \texttt{(signed-byte 32)}.

See the section “Fast 32-bit arithmetic” on page 472 for more information about the INT32 API.
The type `int64` is used to generate optimal 64-bit arithmetic code. Objects of type `int64` are generated and can be manipulated using the functions in the INT64 API but the compiler can optimize such source code by eliminating the intermediate `int64` objects to produce efficient raw 64-bit code. See the section “Fast 64-bit arithmetic” on page 474 for more information.
### Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int64*</td>
<td>The multiply operator for int64 objects.</td>
</tr>
<tr>
<td>int64+</td>
<td>The add operator for int64 objects.</td>
</tr>
<tr>
<td>int64-</td>
<td>The subtract operator for int64 objects.</td>
</tr>
<tr>
<td>int64/</td>
<td>The divide operator for int64 objects.</td>
</tr>
</tbody>
</table>

**Summary**
The arithmetic operators for int64 objects.

**Package**
*system*

**Signatures**
- `int64* x y => int64`
- `int64+ x y => int64`
- `int64- x y => int64`
- `int64/ x y => int64`

**Arguments**
- `x` An int64 object or an integer of type (signed-byte 64).
- `y` An int64 object or an integer of type (signed-byte 64).

**Values**
- `int64` An int64 object.

**Description**
- The function int64* is the multiply operator for int64 objects.
- The function int64+ is the add operator for int64 objects.
- The function int64- is the subtract operator for int64 objects.
- The function int64/ is the divide operator for int64 objects.

See the section “Fast 64-bit arithmetic” on page 474 for more information about the INT64 API.

**See also**
- int64
+int64-0+

Symbol Macro

Summary Shorthand for \((\text{sys:integer-to-int64 } 0)\).

Package system

Description The symbol macro +int64-0+ expands to \((\text{sys:integer-to-int64 } 0)\).

See also integer-to-int64

+int64-1+

Symbol Macro

Summary Shorthand for \((\text{sys:integer-to-int64 } 1)\).

Package system

Description The symbol macro +int64-1+ expands to \((\text{sys:integer-to-int64 } 1)\).

See also integer-to-int64

int64-1+ int64-1-

Functions

Summary The operators for int64 objects corresponding to the functions 1+ and 1-.

Package system

Signatures

\[
\begin{align*}
\text{int64-1+ } x & \Rightarrow \text{int64} \\
\text{int64-1- } x & \Rightarrow \text{int64}
\end{align*}
\]
Arguments

\(x\)  
An \texttt{int64} object or an integer of type \texttt{(signed-byte 64)}.

Values

\texttt{int64}  
An \texttt{int64} object.

Description

The functions \texttt{int64-1+} and \texttt{int64-1-} are the operators for \texttt{int64} objects corresponding to the functions \texttt{1+} and \texttt{1-}.

See the section “Fast 64-bit arithmetic” on page 474 for more information about the INT64 API.

See also \texttt{int64}

\texttt{int64/=}  
\texttt{int64<}  
\texttt{int64<=}  
\texttt{int64=}  
\texttt{int64>}  
\texttt{int64>=}

\textbf{Functions}

Summary

The comparison operators for \texttt{int64} objects.

Package

\texttt{system}

Signatures

\texttt{int64/= x y => result}  
\texttt{int64< x y => result}  
\texttt{int64<= x y => result}  
\texttt{int64= x y => result}  
\texttt{int64> x y => result}  
\texttt{int64>= x y => result}

Arguments

\(x\)  
An \texttt{int64} object or an integer of type \texttt{(signed-byte 64)}. 
An int64 object or an integer of type
(signed-byte 64).

Values

\textit{result} A boolean.

Description

The function int64/= is the not equal comparison for int64 objects.

The function int64< is the less than comparison for int64 objects.

The function int64<= is the less than or equal comparison for int64 objects.

The function int64= is the equal comparison for int64 objects.

The function int64> is the greater than comparison for int64 objects.

The function int64>= is the greater than or equal comparison for int64 objects.

See the section “Fast 64-bit arithmetic” on page 474 for more information about the INT64 API.

See also int64

\textbf{int64<<}

\textbf{int64>>}

\textit{Functions}

Summary

The shift operators for int64 objects.

Package

\textit{system}

Signatures

int64<< x y => result

int64>> x y => result
Arguments

$x$  An int64 object or an integer of type (signed-byte 64).
$y$  An int64 object or an integer of type (signed-byte 64).

Values

$result$  An int64 object.

Description

The function int64<< is a shift left operator for int64 objects.
The function int64>> is a shift right operator for int64 objects.
See the section “Fast 64-bit arithmetic” on page 474 for more information about the INT64 API.

See also

int64

int64-aref

Function

Summary

The accessor for a simple-int64-vector.

Package

system

Signature

int64-aref vector index => int64
(setf int64-aref) x vector index => int64

Arguments

$vector$  An simple-int64-vector.
$index$  A non-negative fixnum.
$x$  An int64 object or an integer of type (signed-byte 64).

Values

$int64$  An int64 object.

Description

The function int64-aref is the accessor for a simple-int64-vector. The reader returns an int64 object for the
value at index \textit{index} in \textit{vector}. The writer sets the value at index \textit{index} in \textit{vector} to the \texttt{int64} object or integer \(x\) supplied. See the section “Fast 64-bit arithmetic” on page 474 for more information about the INT64 API.

See also \texttt{int64\texttt{-\texttt{simple-int64-vector}}}

\texttt{int64-logand}
\texttt{int64-logandc1}
\texttt{int64-logandc2}
\texttt{int64-logbitp}
\texttt{int64-logeqv}
\texttt{int64-logior}
\texttt{int64-lognand}
\texttt{int64-lognor}
\texttt{int64-lognot}
\texttt{int64-logorc1}
\texttt{int64-logorc2}
\texttt{int64-logtest}
\texttt{int64-logxor}  

\textbf{Functions}

\textbf{Summary} The bitwise logical operators for \texttt{int64} objects.

\textbf{Package} \texttt{system}

\textbf{Signatures}

\begin{verbatim}
int64-logand x y => int64
int64-logandc1 x y => int64
int64-logandc2 x y => int64
int64-logbitp index x => result
int64-logeqv x y => int64
\end{verbatim}
The SYSTEM Package

\[\text{int64-logior } x \ y \Rightarrow \text{int64}\]
\[\text{int64-logand } x \ y \Rightarrow \text{int64}\]
\[\text{int64-lognor } x \ y \Rightarrow \text{int64}\]
\[\text{int64-lognot } x \Rightarrow \text{int64}\]
\[\text{int64-logorc1 } x \ y \Rightarrow \text{int64}\]
\[\text{int64-logorc2 } x \ y \Rightarrow \text{int64}\]
\[\text{int64-logtest } x \ y \Rightarrow \text{result}\]
\[\text{int64-logxor } x \ y \Rightarrow \text{int64}\]

**Arguments**

- \(x\): An \text{int64} object or an integer of type \text{(signed-byte 64)}.
- \(y\): An \text{int64} object or an integer of type \text{(signed-byte 64)}.
- \(index\): An \text{int64} object or an integer of type \text{(signed-byte 64)}.

**Values**

- \(\text{int64}\): An \text{int64} object.
- \(\text{result}\): An boolean.

**Description**

The function \text{int64-logand} is the bitwise logical ‘and’ operator for \text{int64} objects.

The function \text{int64-logandc1} is the bitwise logical operator for \text{int64} objects which ‘ands’ the complement of \(x\) with \(y\).

The function \text{int64-logandc2} is the bitwise logical operator for \text{int64} objects which ‘ands’ \(x\) with the complement of \(y\).

The function \text{int64-logbitp} is the test for \text{int64} objects which returns \text{t} if if the bit at index \(index\) in \(x\) is 1, and \text{nil} if it is 0.

The function \text{int64-logeqv} is the bitwise logical operator for \text{int64} objects which returns the complement of the ‘exclusive or’ of \(x\) and \(y\).
The function \texttt{int64-logior} is the bitwise logical ‘inclusive or’ operator for \texttt{int64} objects.

The function \texttt{int64-lognand} is the bitwise logical operator for \texttt{int64} objects which returns the complement of the ‘and’ of \(x \) and \(y\).

The function \texttt{int64-lognor} is the bitwise logical operator for \texttt{int64} objects which returns the complement of the ‘inclusive or’ of \(x \) and \(y\).

The function \texttt{int64-lognot} is the bitwise logical operator for \texttt{int64} objects which returns the complement of its argument \(x\).

The function \texttt{int64-logorc1} is the bitwise logical operator for \texttt{int64} objects which ‘inclusive ors’ the complement of \(x\) with \(y\).

The function \texttt{int64-logorc2} is the bitwise logical operator for \texttt{int64} objects which ‘inclusive ors’ \(x\) with the complement of \(y\).

The function \texttt{int64-logtest} is the bitwise test for \texttt{int64} objects which returns \texttt{t} if any of the bits designated by \(1\) in \(x\) is \(1\) in \(y\), and returns \texttt{nil} otherwise.

The function \texttt{int64-logxor} is the bitwise logical ‘exclusive or’ operator for \texttt{int64} objects.

See the section “Fast 64-bit arithmetic” on page 474 for more information about the INT64 API.

See also \texttt{int64}

\texttt{int64-minusp}  \texttt{int64-plusp}  \texttt{int64-zerop}  

\textit{Summary}  The \texttt{minusp}, \texttt{plusp} and \texttt{zerop} tests for an \texttt{int64} object.
The function \texttt{int64-minus} tests whether its argument \( x \) is \texttt{int64<} than the value of \texttt{+int64-0+}.

The function \texttt{int64-plus} tests whether its argument \( x \) is \texttt{int64>} than the value of \texttt{+int64-0+}.

The function \texttt{int64-zerop} tests whether its argument \( x \) is \texttt{int64=} to the value of \texttt{+int64-0+}.

See the section “Fast 64-bit arithmetic” on page 474 for more information about the INT64 API.

See also \texttt{int64}

\textbf{Function} \texttt{int64-to-int32}

\textbf{Summary} Converts from \texttt{int64} to \texttt{int32}.

\textbf{Package} \texttt{system}

\textbf{Signature} \texttt{int64-to-int32} \( x \Rightarrow y \)

\textbf{Arguments} \( x \) An \texttt{int64} object.

\textbf{Values} \( y \) An \texttt{int32} object.
The function `int64-to-int32` converts the `int64` object $x$ to the corresponding `int32` object $y$.

See also

- `int32`
- `int32-to-int64`
- `int64`

### int64-to-integer

**Function**

**Summary**
The destructor converting an `int64` object to an integer.

**Package**
`system`

**Signature**
`int64-to-integer int64 => integer`

**Arguments**
`int64` An `int64` object or an integer of type `(signed-byte 64)`.

**Values**
`integer` An integer of type `(signed-byte 64)`.

**Description**
The function `int64-to-integer` returns an integer `integer` of type `(signed-byte 64)` corresponding to the `int64` object `int64`. The argument `int64` can also be an integer of type `(signed-byte 64)`, in which case it is simply returned.

An error is signaled if `int64` is not of type `int64` or `(signed-byte 64).

See the section “Fast 64-bit arithmetic” on page 474 for more information about the INT64 API.

See also

- `int64`

### integer-to-int32

**Function**

**Summary**
The constructor for `int32` objects.
The `integer-to-int32` function constructs an `int32` object from an integer. An error is signaled if the integer is not of type `(signed-byte 32)`. See the section “Fast 32-bit arithmetic” on page 472 for more information about the INT32 API.

See also `int32`

---

The `integer-to-int64` function constructs an `int64` object from an integer. An error is signaled if the integer is not of type `(signed-byte 64)`. See the section “Fast 64-bit arithmetic” on page 474 for more information about the INT64 API.

See also `int64`
**line-arguments-list**  

*Variable*

**Summary**
List of the command line arguments used when LispWorks was invoked.

**Package**
`system`

**Initial value**
`nil`

**Description**
This variable contains a list of strings. These are the arguments with which LispWorks was called, in the same order. The first element is the executable itself.

You can implement command line processing in your application by testing elements in `*line-arguments-list*`. Use a string comparison function such as `string=` to compare them.

For a description of the command line arguments processed by LispWorks, see “The Command Line” on page 454.

**See also**
`lisp-image-name`

---

**load-data-file**  

*Function*

**Summary**
Loads a binary data file created by `dump-forms-to-file` or `with-output-to-fasl-file`.

**Package**
`system`

**Description**
`load-data-file` is now exported from the HCL package, and has additional functionality since LispWorks 7.0

View the full manual entry for details: `load-data-file`. 
locale-file-encoding

Summary
Provides an encoding corresponding to the current code page on Microsoft Windows, and the locale on Unix.

Package
system

Signature
locale-file-encoding pathname ef-spec buffer length => new-ef-spec

Arguments
pathname
Pathname identifying location of buffer.

ef-spec
An external format spec.

buffer
A buffer whose contents are examined.

length
Length (an integer) up to which buffer should be examined.

Values
new-ef-spec
Default external format spec created by merging ef-spec with the encoding that was found.

Description
The function locale-file-encoding consults the ANSI code page on Microsoft Windows. If the code page identifier is in win32:*latin-1-code-pages*, locale-file-encoding merges ef-spec with :latin-1. This external format writes Latin-1 on output, giving an error for any non-Latin-1 characters that are written. If the code page identifier is not in win32:*latin-1-code-pages* then locale-file-encoding merges ef-spec with an encoding corresponding to the current code page that gives an error for characters that cannot be encoded.

locale-file-encoding merges ef-spec with :latin-1 on Unix.

See also
*file-encoding-detection-algorithm*
*latin-1-code-pages*
## low-level-atomic-place-p

### Function

**Summary**
The predicate for whether a place is suitable for use with the low-level atomic operators.

**Signature**

```
low-level-atomic-place-p place optional environment => result
```

**Arguments**

- `place`: A place
- `environment`: An environment object

**Values**

- `result`: A boolean

**Description**

The function `low-level-atomic-place-p` is the predicate for whether the place `place` is one of the places for which low-level atomic operations are defined, and is therefore suitable for use with those operators.

These places are described in "Low level atomic operations" on page 299.

**See also**

- `atomic-decf`
- `atomic-exchange`
- `atomic-fixnum-decf`
- `atomic-pop`
- `atomic-push`
- `compare-and-swap`
- `define-atomic-modify-macro`
**make-current-allocation-permanent**

**Function**

**Summary**

Mobile GC only: Makes all the objects currently in generation 2 permanent (non-GCable).

**Package**

system

**Signature**

make-current-allocation-permanent &key gc-p coalesce

**Arguments**

`gc-p`, `coalesce`

Booleans.

**Description**

The function **make-current-allocation-permanent** makes all the objects currently allocated in generation 2 permanent, which means that they will never be GCed.

If `gc-p` is non-nil (the default) then **make-current-allocation-permanent** does an initial GC by calling `(gc-generation 2 :coalesce coalesce)`, which by default means that all currently live objects are promoted to generation 2, and hence are made permanent. `coalesce` defaults to `t`. See the documentation for `gc-generation` for details.

For Static objects, only segments that are in generation 2 are made permanent (because static objects are never promoted between generations).

The function **generation-number** returns 3 when its argument is a permanent object. `room` reports the Other and Cons objects that were made permanent under Permanent Other and Permanent Cons, except Large and Static, where `(room t)` reports permanent segments as being in generation 3.

Making objects permanent saves work for the GC, but wastes some memory. Repeated calls to **make-current-allocation-permanent** wastes more memory. The operation itself is fast, but the initial GC takes time depending on the amount allocated.
Notes

The operation is done by moving whole segments to the permanent segments, which means that any free area in the segments is moved as well and hence is wasted (permanently). It is therefore essential to reduce the free area in generation 2 before calling `make-current-allocation-permanent` by performing a GC of generation 2. Hence you should pass `gc-p nil` only if you already did the GC of generation 2 explicitly.

Passing `coalesce nil` means that currently live objects in generation 0 are not made permanent. This is useful for objects that are short-lived, but will cause young long-lived objects to stay in the GCed generations. The effect either way is unlikely to be large.

Note also that since permanent objects are not GCed, a permanent object that points to a non-permanent one will keep the non-permanent object live forever (unless the pointer is overwritten explicitly by the application). That will make the non-permanent object live forever as well, and hence add work for the GC.

The main effect of making objects permanent is to reduce the time and the memory peak required for GC of generation 2, so can have a very beneficial effect on performance. It is particularly useful if the relatively few objects are allocated after the call that live forever, so the size of generation 2 after a GC of generation 2 is relatively small. Using `make-current-allocation-permanent` is probably useful even if 20% of the permanent objects would have died after a while if left in the GCable generations. If the application does not create new permanent objects, but does have objects that live long enough to be promoted to generation 2 before dying ("generation leak"), it maybe useful to call `make-current-allocation-permanent` even if 50% of the objects would have died otherwise. These percentages should only be used as a guide.

`make-current-allocation-permanent` wastes the memory that is free in generation 2 before the operation. The amount of free memory after the initial GC is typically independent
of the amount allocated, and averages around 8 MB. Thus it is not useful to use `make-current-allocation-permanent` unless you have significantly more than 8 MB of permanent objects. The waste happens on each call to `make-current-allocation-permanent`, so you should minimize the number of calls and typically call it once in a run of the application.

The amount wasted in the permanent areas is the amount that `room` reports as free under Permanent Cons and Permanent Other, plus the size of the objects in these areas that are effectively dead (not pointed by any other live object). Since the GC does not collect the permanent objects, there is no easy way to know which of them are effectively dead. If you want to know that, you need to run the application without calling `make-current-allocation-permanent`, see how much is allocated in generation 2 in this case, and compare this to the amount allocated permanently when you do call `make-current-allocation-permanent`.

Large objects (currently that means larger than 1 MB) can be made permanent individually by `make-object-permanent` and `make-permanent-simple-vector`, and can be explicitly released and the memory returned to the operating system by using `release-object-and-nullify`.

See also

- `gc-generation`
- `make-object-permanent`
- `make-permanent-simple-vector`
- `release-object-and-nullify`
- “Preventing/reducing GC of generation 2” on page 159
- “Mobile GC technical details” on page 154

**Function**

`make-gesture-spec`

**Summary** Create a Gesture Spec object.
Package  

Signature  
make-gesture-spec data modifiers &optional can-shift-both-case-p => gspec

Arguments  
data  A non-negative integer less than cl:char-code-limit, or a Gesture Spec keyword, or nil.

modifiers  A non-negative integer less than 64, or nil.

can-shift-both-case-p  A generalized boolean.

Values  
gspec  A Gesture Spec object

Description  
The function make-gesture-spec returns a new Gesture Spec object gspec. This can be used to represent a keystroke consisting of the key indicated by data, modified by the modifier keys indicated by modifiers.

If data is an integer, it represents the key (code-char data).
If data is a keyword, it must be one of the known Gesture Spec keywords and represents the key with the same name. If data is nil, then gspec has a wild data component.

These are the Gesture Spec keywords:

:f1
:f2
:f3
:f4
:f5
:f6
:f7
:f8
:f9
:f10
The SYSTEM Package

...
:prior
:next
:end
:begin
:select
:print
:execute
:insert
:undo
:redo
:menu
:find
:cancel
:break
:clear
:pause
:kp-f1
:kp-f2
:kp-f3
:kp-f4
:kp-enter
:applications-menu
:print-screen
:scroll-lock
:sys-req
:reset
:stop
:user
:system
:clear-line
:clear-display
Not all of these Gesture Spec keywords will be generated by all platforms and/or keyboards.

If `modifiers` is an integer, it represents modifier keys according to the values `gesture-spec-accelerator-bit`, `gesture-spec-control-bit`, `gesture-spec-hyper-bit`, `gesture-spec-meta-bit`, `gesture-spec-shift-bit`, and `gesture-spec-super-bit`. If `modifiers` is `nil`, then `gspec` has a wild modifiers component.

The gesture `Shift+X` could potentially be represented by the unmodified uppercase character `X`, or lowercase `x` with the `Shift` modifier. In order to ensure a consistent representation the latter form is not supported by Gesture Specs by default. That is, a `both-case-p` character may not be combined with the single modifier `Shift` in the accelerator argument. This can be overridden by passing a true value for `can-shift-both-case-p`.

A `both-case-p` character is allowed with `Shift` if there are other modifiers. See the below for examples.

Wild Gesture Specs can be useful when specifying an input model for a `capi:output-pane`.

**Example**

```lisp
(sys:make-gesture-spec
  97
  (logior sys:gesture-spec-control-bit
           sys:gesture-spec-meta-bit))
```

A `both-case-p` character may not be combined with the single modifier `Shift` in the accelerator argument, so code like this signals an error:
(sys:make-gesture-spec  
(char-code #\x)  
sys:gesture-spec-shift-bit)

Instead you should use:

(sys:make-gesture-spec (char-code #\X) 0)

A both-case-p character is allowed with Shift if there are other modifiers:

(sys:make-gesture-spec  
(char-code #\x)  
(logior sys:gesture-spec-shift-bit sys:gesture-spec-meta-bit))

See also gesture-spec-accelerator-bit
gesture-spec-control-bit
gesture-spec-data
gesture-spec-hyper-bit
gesture-spec-meta-bit
gesture-spec-modifiers
gesture-spec-p
gesture-spec-shift-bit
gesture-spec-super-bit
print-pretty-gesture-spec

make-object-permanent

Function

Summary Mobile GC only: Make a large object permanent.

Package system

Signature make-object-permanent object => did-it-p

Arguments object An object that is allocated in its own segment.

Values did-it-p A boolean.
Description

The function `make-object-permanent` makes `object` permanent (if possible), which means that GC will never scan or free it (but will still follow pointers from it). That reduces the amount of work for the GC.

`make-object-permanent` can only make `object` permanent if it is allocated in its own segment, so it must be a large object (> 1 MB).

`make-object-permanent` returns true if `object` was made permanent (or is already permanent) and false otherwise.

Notes

An object that has been made permanent will never be freed by the GC, so you must use `release-object-and-nullify` to free it.

After the object is made permanent, the segment in which the object resides is reported by `(room t)` to be in generation 3.

`make-object-permanent` does not work (it just returns false) on an array that is displaced to a vector that is allocated on its own segment. To work it must be called on the vector itself.

Large vectors that do not contain pointers (that is every vector except `simple-vector`) are not scanned by the GC, so making them permanent does not give a significant gain. Thus `make-object-permanent` is really useful only for `simple-vector` objects.

If you make a new large `simple-vector` objects and want to make them permanent immediately, it is better to use `make-permanent-simple-vector`, because `make-object-permanent` causes the next GC to take more time, while `make-permanent-simple-vector` does not (unless supplied an initial-element which is not immediate or permanent).

See also

`make-permanent-simple-vector`
`release-object-and-nullify`
`allocated-in-its-own-segment-p`

“Special considerations for the Mobile GC” on page 157
**Function**

**make-permanent-simple-vector**

**Summary**
Create a permanent (when possible) `simple-vector`.

**Package**
`system`

**Signature**

```lisp
(make-permanent-simple-vector size &optional initial-element) => simple-vector
```

**Arguments**

- `size` A fixnum.
- `initial-element` Any Lisp object.

**Values**

- `simple-vector` A `simple-vector`.

**Description**

The function `make-permanent-simple-vector` creates a `simple-vector` of length `size` with initial element `initial-element` as if by the call `(make-array size :initial-element initial-element)`, except that it is allocated as a permanent object when possible.

When not in the Mobile GC, `simple-vector` is allocated in the highest generation number.

In the Mobile GC, if `size` is larger than `(ash 1 17)` (#+20000, 131072), so `simple-vector` is allocated in its own segment, it is made permanent. Otherwise it is allocated n generation 2.

**Notes**

`make-permanent-simple-vector` is intended mainly for allocating large `simple-vector` objects in the Mobile GC (that is, those that can be made permanent). When not in the Mobile GC, it does not do anything that `make-array` cannot do, but it may be convenient sometimes.

Note that, except for large `simple-vector` objects in the Mobile GC, `simple-vector` is not actually permanent, and a GC of the highest generation will scan it (or free it if nothing point to it).
When `simple-vector` is permanent, and you do not need it any more, then you need to release it by `release-object-and-nullify`.

In the Mobile GC with large vectors, if `initial-element` is not supplied or it is an immediate or a permanent object, `make-permanent-simple-vector` is much better than using `make-object-permanent` after a call to `make-array`, because it knows that it does not contain pointers to a lower generation.

See also
- `make-object-permanent`
- `release-object-and-nullify`
- `allocated-in-its-own-segment-p`
- “Special considerations for the Mobile GC” on page 157

**make-simple-int32-vector**

**Function**

**Summary**
The constructor for `simple-int32-vector` objects.

**Package**
`system`

**Signature**

`make-simple-int32-vector length &key initial-contents initial-element => vector`

**Arguments**

- `length` A non-negative fixnum.
- `initial-contents` A sequence of integers of type `(signed-byte 32)`, or `nil`.
- `initial-element` An integer of type `(signed-byte 32)`.

**Values**

- `vector` A `simple-int32-vector`.

**Description**
The function `make-simple-int32-vector` is the constructor for `simple-int32-vector` objects.

The argument `initial-contents`, if supplied, should be a sequence of length `length`. It specifies the contents of `vector`. 
The argument *initial-element*, if supplied, specifies the contents of *vector*.

An error is signaled if both *initial-contents* and *initial-element* are supplied.

See the section “Fast 32-bit arithmetic” on page 472 for more information about the INT32 API.

See also

- int32
- simple-int32-vector

**make-simple-int64-vector**

*Function*

**Summary**

The constructor for *simple-int64-vector* objects.

**Package**

*system*

**Signature**

`make-simple-int64-vector length &key initial-contents initial-element => vector`

**Arguments**

- *length*  
  A non-negative fixnum.
- *initial-contents*  
  A sequence of integers of type `(signed-byte 64)`, or nil.
- *initial-element*  
  An integer of type `(signed-byte 64)`.

**Values**

- *vector*  
  A *simple-int64-vector*.

**Description**

The function *make-simple-int64-vector* is the constructor for *simple-int64-vector* objects.

The argument *initial-contents*, if supplied, should be a sequence of length *length*. It specifies the contents of *vector*.

The argument *initial-element*, if supplied, specifies the contents of *vector*. 
An error is signaled if both initial-contents and initial-element are supplied.

See the section “Fast 64-bit arithmetic” on page 474 for more information about the INT64 API.

See also

- int64
- simple-int64-vector

### make-stderr-stream

**Function**

**Summary**

Returns an output stream connected to stderr.

**Package**

system

**Signature**

make-stderr-stream => stream

**Arguments**

None.

**Values**

- stream  An output stream.

**Description**

The function make-stderr-stream returns an output stream connected to stderr.

make-stderr-stream returns the same stream each time. Calling close on this stream has no effect (except that it forces the output).

**Notes**

1. On Microsoft Windows, if the stderr is not redirected on the command line then output to the stderr stream appears in a console.

   The console window will be created if it does not exist. That is not desirable for typical (non-console) applications. Therefore writing to the stderr stream is probably useful only in a console application (see the :console keyword argument in save-image), or when you know that stderr is going to be redirected.
2. Ensure your delivered Windows application calls `make-stderr-stream` at run time rather than in the build script, because it contains the handle of Windows stderr.

3. On Mac OS X, applications that are launched from the desktop have their stderr redirected to the "console messages".

### make-typed-aref-vector

**Function**

**Summary**
Makes a vector that can be accessed efficiently.

**Package**
`system`

**Signature**
`make-typed-aref-vector byte-length => vector`

**Arguments**
`byte-length` A non-negative fixnum.

**Values**
`vector` A vector.

**Description**
The function `make-typed-aref-vector` returns a vector which is suitable for efficient access at compiler optimization level safety = 0.

`byte-length` is measured in 8-bit bytes.

Use `typed-aref` to access `vector` efficiently.

**Notes**
Declaring the result of `make-typed-aref-vector` as `cl:dynamic-extent` causes it to allocate the array on the stack (in LispWorks 7.0 and later versions).

**Examples**
To make a typed vector of the type `(unsigned-byte 32)` or `single-float` with length 10:

`(make-typed-aref-vector (* 10 4))`

To make a typed vector of the type `double-float` with length 10:
(make-typed-aref-vector (* 10 8))

See also typed-aref
“Optimized integer arithmetic and integer vector access” on page 472

map-environment

Function

Summary
Maps functions over the bindings in an environment.

Package
system

Signature
map-environment env &key variable function block tag

Arguments
env An environment or nil
variable A function designator
function A function designator
block A function designator
tag A function designator

Description
The function map-environment calls variable for each local variable binding in env, function for each local function binding in env, block for each block binding in env and tag for each tag binding in env.

variable is called with the following arguments: name kind info
name A symbol naming a variable
kind One of :special, :symbol-macro or :lexical, which specifies the kind of binding (see variable-information)
info The symbol-macro expansion if kind is :symbol-macro and is unspecified otherwise.
function is called with the following arguments: name kind info
name A symbol naming a function.
kind One of :macro or :function, which specifies the kind of binding (see function-information).
info The macro expansion function if kind is :macro and is unspecified otherwise.

block is called with the following arguments: name kind info
name A symbol naming a block.
kind The keyword :block
info Unspecified.
tag is called with the following arguments: name kind info
name A symbol naming a tag.
kind The keyword :tag
info Unspecified

See also augment-environment
declaration-information
define-declaration
function-information
variable-information

marking-gc Function
Summary Performs a Marking GC in 64-bit LispWorks.
Package system
Signature marking-gc gen-num &key what-to-copy max-size max-size-to-copy fragmentation-threshold
Arguments gen-num An integer in the inclusive range [0,7].
The function `marking-gc` garbage collects (GCs) the generation specified by `gen-num`, and all younger generations. It uses mark and sweep, rather than copy.

Mark and sweep garbage collection uses less virtual memory during its operation, but leaves the memory fragmented, which has a detrimental effect on the performance of the system afterwards. It is therefore not used automatically by the system, except to garbage collect static objects.

`marking-gc` is useful when you want to GC a generation which contains large amount (gigabytes) of data, to make sure there are no spurious pointers from this generation to a younger generation, and you do not expect many objects in the large generation to be collected. In this scenario, a Copying GC would use virtual memory which is almost double the size of the large generation during its operation, and so would possibly cause heavy paging.

Marking GC causes fragmentation. You can reduce the amount of fragmentation by supplying either (or both) of the arguments `what-to-copy` and `max-size-to-copy`. These specify that part of the data should be collected by copying instead. Using some copying GC rather than mark and sweep will reduce the amount of fragmentation.

`what-to-copy` specifies the allocation type to copy. It can be one of the main allocation types or `:weak`, meaning copy only objects in segments of that type. `what-to-copy` can also be `:all`, meaning copy objects in all segments. If `what-to-copy` is

### what-to-copy
One of the keywords `:cons`, `:symbol`, `:function`, `:non-pointer`, `:other`, `:weak`, `:all` or `:default`.

### max-size-to-copy
A positive number or `nil`.

### max-size
A synonym for `max-size-to-copy`.

### fragmentation-threshold
A number in the inclusive range `[0, 10]`. Description

The function `marking-gc` garbage collects (GCs) the generation specified by `gen-num`, and all younger generations. It uses mark and sweep, rather than copy.

Mark and sweep garbage collection uses less virtual memory during its operation, but leaves the memory fragmented, which has a detrimental effect on the performance of the system afterwards. It is therefore not used automatically by the system, except to garbage collect static objects.

`marking-gc` is useful when you want to GC a generation which contains large amount (gigabytes) of data, to make sure there are no spurious pointers from this generation to a younger generation, and you do not expect many objects in the large generation to be collected. In this scenario, a Copying GC would use virtual memory which is almost double the size of the large generation during its operation, and so would possibly cause heavy paging.

Marking GC causes fragmentation. You can reduce the amount of fragmentation by supplying either (or both) of the arguments `what-to-copy` and `max-size-to-copy`. These specify that part of the data should be collected by copying instead. Using some copying GC rather than mark and sweep will reduce the amount of fragmentation.

`what-to-copy` specifies the allocation type to copy. It can be one of the main allocation types or `:weak`, meaning copy only objects in segments of that type. `what-to-copy` can also be `:all`, meaning copy objects in all segments. If `what-to-copy` is
:default then each call to marking-gc chooses one of the main allocation types or :weak to copy, and successive calls with :default cycle through these allocation types.

max-size-to-copy can be used to limit the amount that is copied, and thus limit the virtual memory that the operation needs. If max-size-to-copy is non-nil, it specifies the limit, in gigabytes, of memory that can be used for copying. If there is more than max-size-to-copy gigabytes of data of the type what-to-copy, the rest of this data is garbage collected by marking. The default value of max-size-to-copy is nil, which means there is no limit on the amount that is copied.

fragmentation-threshold should be a number between 0 and 10. It specifies a minimum ratio between the free area in a segment that cannot be easily used for more allocation and the allocated area in this segment. Segments that are below this threshold are not copied. The default value of fragmentation-threshold is 1.

Notes  
marking-gc is implemented only in 64-bit LispWorks. It is not relevant to the Memory Management API in 32-bit implementations.

In the Mobile GC, marking-gc is equivalent to (gc-generation gen-num).

See also  
gc-generation  
set-blocking-gen-num  
“Guidance for control of the memory management system” on page 130

memory-growth-margin

Function

Summary  
Returns the difference between the top of the Lisp heap and a maximum memory limit in 32-bit LispWorks.

Package  
system
The SYSTEM Package

Signature  
memory-growth-margin => result

Values  
result  An integer address, or nil.

Description  
If a limit on the maximum memory has been set by set-maximum-memory, then memory-growth-margin returns the difference between the current top of the Lisp heap and that limit. That is, the amount by which the heap can grow. Otherwise memory-growth-margin returns nil. This is the default behavior.

Notes  
memory-growth-margin is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations.

See also  
set-maximum-memory
“Memory Management in 32-bit LispWorks” on page 135

merge-ef-specs  

Function

Summary  
Creates a new external format spec from two other external format specs.

Package  
system

Signature  
merge-ef-specs ef-spec1 ef-spec2 => ef-spec

Arguments  
ef-spec1  An external format spec.

ef-spec2  An external format spec.

Values  
ef-spec  The resultant external format spec created from information in ef-spec1 and ef-spec2.
The function `merge-ef-specs` returns an external format spec constructed by adding information not supplied in `ef-spec1` from `ef-spec2`.

Each external format spec argument is either a symbol or a list.

If `ef-spec1` and `ef-spec2` have the same value for their name component (whether they are lists or symbols), return `ef-spec1` combined with any parameters from `ef-spec2` that are not specified in `ef-spec1`.

Otherwise, if `ef-spec1` is `:default` or a list beginning with `:default`, return `ef-spec2` with parameters modified to be a union of the parameters from `ef-spec1` and `ef-spec2`, with those from `ef-spec1` taking priority.

Otherwise, return `ef-spec1` with any `:eol-style` parameter from `ef-spec2` if `ef-spec1` does not specify `:eol-style`.

### mobile-gc-p

**Function**

**Summary**

Returns true when using the Mobile GC, otherwise false.

**Package**

`system`

**Signature**

`mobile-gc-p \Rightarrow result`

**Values**

`result` A boolean.

**Description**

The function `mobile-gc-p` is a predicate that returns true when 64-bit LispWorks is using the Mobile GC and false otherwise. In 32-bit LispWorks, it always returns false.

**See also**

“The Mobile GC” on page 151
**mobile-gc-sweep-objects**  Function

**Summary**
Mobile GC only: Sweeps objects for the Mobile GC.

**Package**
*system*

**Signature**
`mobile-gc-sweep-objects function &key permanent permanent-weak permanent-non-pointer permanent-new long-lived static large  
gen-0  gen-1  gen-2`

**Arguments**
- `function`: A function or fbound symbol that takes one argument.
- `permanent`, `permanent-weak`, `permanent-non-pointer`, `permanent-new`, `long-lived`, `static`, `large`, `gen-0`, `gen-1`, `gen-2`: Booleans.

**Description**
The function `mobile-gc-sweep-objects` sweeps objects in the manner that `sweep-all-objects` and `sweep-gen-num-objects` do, but gives you a better control over which objects are swept. It is therefore most efficient for sweeping certain subsets of the objects.

`function` is called with each object that matches the criteria specified by non-nil values of the keyword arguments.

Permanent objects are objects that the GC does not scan or free, which include objects that were alive when the image was delivered and objects that were alive when `make-current-allocation-permanent` was called. Large objects that were made permanent by `make-object-permanent` or `make-permanent-simple-vector` are also permanent, but in `mobile-gc-sweep-objects` sweeping them is controlled by `large`.

`permanent` defaults to `nil`. It controls whether sweeping includes permanent objects that contain pointers, and is also the default value for `permanent-weak`, `permanent-non-pointer` and `permanent-new`. `permanent-weak` controls sweeping of per-
manent weak objects. *permanent-non-pointer* control sweeping of permanent objects that do not contain pointers. *permanent-new* controls sweeping of objects that were made permanent by a call to `make-current-allocation-permanent`.

`long-lived` defaults to `nil`, and is used as the default value for `gen-2`, `gen-1`, `large` and `static`. `gen-0` defaults to `nil`. Therefore, if `long-lived` is non-nil and no other keyword arguments are supplied, then all non permanent objects are swept except those in generation 0.

`gen-2`, `gen-1`, `gen-0`, `large` and `static` control the sweeping of ordinary objects in generation 2, generation 1, generation 0, large objects and static objects respectively.

**Notes**

With the default values of the keywords, `mobile-gc-sweep-objects` does nothing.

`mobile-gc-sweep-objects` is useful for sweeping specific objects, for example all static objects. For this case, it is much more efficient than using `sweep-all-objects` and checking each object using `staticp`.

`mobile-gc-sweep-objects` is thread-safe but not atomic with respect to allocation or GC, so gets confused if a GC occurs while it is sweeping segments that are affected by the GC or there is allocation in any of these segments. It is therefore rarely useful to sweep generation 0, and sweeping generation 1 is probably not useful either.

**See also**

`sweep-all-objects`

`sweep-gen-num-objects`

“Mobile GC technical details” on page 154

---

**object-address**

**Function**

**Summary**

Returns the address of a Lisp object.
### object-address

<table>
<thead>
<tr>
<th>Package</th>
<th>system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>object-address object =&gt; address</td>
</tr>
<tr>
<td>Arguments</td>
<td>object</td>
</tr>
<tr>
<td>Values</td>
<td>address</td>
</tr>
<tr>
<td>Description</td>
<td>The function object-address returns the address of the Lisp object object as an integer address. Note that the address of object may change during garbage collection so this integer should be used for debugging purposes only.</td>
</tr>
<tr>
<td>See also</td>
<td>immediatep</td>
</tr>
<tr>
<td></td>
<td>object-pointer</td>
</tr>
<tr>
<td></td>
<td>pointer-from-address</td>
</tr>
</tbody>
</table>

### object-pointer

**Function**

<table>
<thead>
<tr>
<th>Package</th>
<th>system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>object-pointer object =&gt; result</td>
</tr>
<tr>
<td>Arguments</td>
<td>object</td>
</tr>
<tr>
<td>Values</td>
<td>result</td>
</tr>
<tr>
<td>Description</td>
<td>The function object-pointer returns an integer specifying the actual representation of the object object. For most objects, that would be the pointer to it, which is its address (as returned by object-address) plus some tag. Some objects are &quot;immediate&quot; (that is they do not use memory, and immediatep returns t) and for these object-pointer returns the actual address.</td>
</tr>
</tbody>
</table>
The Garbage Collector can move objects, therefore the result of \texttt{object-pointer} is not permanent. It should be used only for debugging.

Notes

The result of \texttt{object-pointer} is what \texttt{cl:print-unreadable-object} uses for the object’s “identity”. It is normally what appears when using \texttt{cl:print-unreadable-object} with \texttt{identity t}.

Examples

\begin{verbatim}
(let ((gf #'make-instance))
  (format t "-a pointer is -x-%s" gf
          (sys:object-pointer gf)))
\end{verbatim}

See also

\texttt{immediatep}, \texttt{object-address}, \texttt{pointer-from-address}

\section*{octet-ref}

\section*{base-char-ref}

\textbf{Functions}

\textbf{Summary}

Loads an octet from a simple vector and returns it as an integer or \texttt{base-char}.

\textbf{Package}

\texttt{system}

\textbf{Signature}

\begin{verbatim}
octet-ref vector octet-index => iresult
(setf octet-ref) int vector octet-index => int
base-char-ref vector octet-index => cresult
(setf base-char-ref) char vector octet-index => char
\end{verbatim}

\textbf{Arguments}

\begin{verbatim}
vector A simple-base-string, a simple-bmp-string or a simple binary vector meaning a vector of element type (unsigned-byte n) or (signed-byte n) for n = 8, 16, 32. In 64-bit LispWorks, n = 64 is also supported.
\end{verbatim}
**octet-index**  A non-negative integer.

**int**  An integer.

**char**  A `base-char`.

**Values**

**iresult**  An integer in the inclusive range $[0, 255]$.

**cresult**  A `base-char`.

**Description**

The functions `octet-ref` and `load` an octet (8-bits element) from the simple vector `vector` at offset `octet-index`, and return it as an integer or `base-char`.

`vector` must be either a string with element type `base-char` or `bmp-char` or a binary vector (as defined above). `vector` cannot be displaced, adjustable or have a fill pointer, and it cannot be a string with element type `character`.

`octet-index` must be an integer, which is used as the count of octets (rather than array elements) to compute the actual offsets.

`octet-ref`, `base-char-ref` and their setters are intended to allow efficient access to `(unsigned-byte 8)` vectors and `simple-base-string` in the same code. For these types of vector they match what `aref` and `(setf aref)` do except that they always take and return the same value/result type, while `aref` and `(setf aref)` take and return a value of a type which depends on the type of the vector. `octet-ref` (and `base-char-ref`) are also more efficient than `aref`.

`octet-ref`, `base-char-ref` and their setters also work on simple binary vectors with element length other than 8 bits, and the results are consistent between themselves. However their results for such vectors do not match `aref`, because they will load and set either part of an element or multiple elements. Also the results of `octet-ref` (and `base-char-ref`) and the result of `aref` can differ between different platforms due to endianness.
Notes: octet-ref, base-char-ref and their setters cannot be used on a simple-text-string.

See also: “Optimized integer arithmetic and integer vector access” on page 472

open-pipe

Function

Summary: Runs an executable or shell command in a subshell.

Package: system

Signature: open-pipe command &key direction element-type interrupt-off shell-type use-pty save-exit-status => stream

Arguments:
- command: A string, a list of strings, a simple-vector of strings, or nil.
- element-type: A type specifier.
- shell-type: A shell type.
- use-pty: A boolean.
- save-exit-status: A boolean.

Values:
- stream: A pipe stream.

Description:
On non-Windows platforms the behavior of open-pipe is analogous to that of popen in the UNIX library. It creates a pipe to/from a subprocess and returns a stream. The stream can be read from or written to as appropriate.

On Microsoft Windows open-pipe calls CreateProcess and CreatePipe and returns a bidirectional stream.
command is interpreted as by `call-system-showing-output`.

direction is a keyword for the stream direction. The default value is `:input`. Bidirectional (I/O) pipes may be created by passing `:io`. See the example below. direction can also be `:none`, which means no input and no output like `call-system`, but is useful when you want to use `pipe-exit-status` and `pipe-kill-process`. On Windows it is not possible to open a unidirectional pipe, so `:input` and `:output` both have the same effect as `:io`.

When `save-exit-status` is non-nil, the status of the child process that `open-pipe` creates is tracked, so `pipe-exit-status` and `pipe-kill-process` can be used reliably. The default value of `save-exit-status` is `nil`.

element-type specifies the type of the stream as with `open`. The default value is `base-char`. This argument is ignored on Microsoft Windows.

`interrupt-off`, if `t`, ensures that Ctrl+C (SIGINT) to the LispWorks image is ignored by the subprocess. This argument is not implemented on Microsoft Windows.

`shell-type` specifies the type of shell to run. On Unix-like systems the default value is "/bin/sh". On Microsoft Windows the default value is "cmd".

`use-pty` is useful on Unix-like systems if the sub-process behaves differently when running interactively and non-interactively. When `use-pty` is non-nil, the input and output of the sub-process are opened using PTY (pseudo-pty). That means that the sub-process sees its input and output as if they come from an interactive terminal. The PTY also processes special characters such as Ctrl-C the same way that an ordinary TTY does.

`use-pty` is probably not useful on Microsoft Windows as there is no concept corresponding to the Unix behavior. If `use-pty` is non-nil then it uses the `CREATE_NEW_PROCESS_GROUP` flag.
when creating the child, but it is not obvious when this might be useful.

stream supports mixed character and binary I/O in the same way as file streams constructed by open.

Examples

Example on Unix:

CL-USER 1 > (setf *ls* (sys:open-pipe "ls"))
Warning: Setting unbound variable *LS*
#<SYSTEM::PIPE-STREAM "ls">

CL-USER 2 > (loop while
  (print (read-line *ls* nil nil)))

"hello"
"othello"
NIL
NIL

CL-USER 3 > (close *ls*)
T

The following example shows you how to use bidirectional pipes.

CL-USER 1 > (with-open-stream
  (s (sys:open-pipe "/bin/csh"
    :direction :io))
  (write-line "whereis ls" s)
  (force-output s)
  (read-line s))

"ls: /sbin/ls /usr/bin/ls /usr/share/man/man1.Z/ls.1"
NIL

Example on Microsoft Windows
This last example illustrates the use of `save-exit-status`. This form runs LispWorks as a subprocess such that it quits immediately with exit status 1623:

```lisp
(setq *sub* (sys:open-pipe (list (lisp-image-name) "-eval" "(quit :status 1623)" :save-exit-status t))

This form then returns 1623:

```lisp
(sys:pipe-exit-status *sub*)

See also
``` lisp
call-system
call-system-showing-output
pipe-exit-status
pipe-kill-process
pipe-close-connection
open-url

Summary Displays a HTML page in a web browser.

Package system

Signature open-url url

Arguments url A string.

Description The function open-url displays the page at the URL url in a web browser.

Supported browsers are Netscape, Firefox, Mozilla, Opera on all platforms, Microsoft Internet Explorer on Microsoft Windows and Mac OS X, plus Safari on Mac OS X.

open-url is defined in the "hqn-web" module.

Compatibility notes If your code uses the unsupported function hqn-web:browse please change to use open-url in LispWorks 5.0 and later.

Examples (sys:open-url "www.lispworks.com")

See also *browser-location*

package-flagged-p

Summary Queries whether a package is flagged.

Package system

Signature package-flagged-p package flag => result

Arguments package A package designator.
flag A keyword.
The function `package-flagged-p` is the predicate for whether the package `package` is flagged with the keyword `flag`.

Current valid values for `flag` are:

- `:implementation` Packages that are part of the LispWorks implementation. You must not add definitions to them.
- `:documented` Packages that are fully documented (that is, all external symbols are documented), and all external symbols are intended for your use.

**pipe-close-connection**

**Summary**
Close the connection that a pipe-stream uses without closing the stream.

**Package**
`system`

**Signature**
`pipe-close-connection stream`

**Arguments**
- `stream` A pipe stream.

**Description**
The function `pipe-close-connection` closes the connection underlying `stream` without closing the stream itself. This means that you cannot communicate with the child process anymore, but `pipe-exit-status` can still return the exit-status of the child process after a call to `pipe-close-connection`. This differs from `close`, which prevents `pipe-exit-status` from working on Microsoft Windows. You should still call `close` on `stream` when you have finished using it.
Notes: `pipe-close-connection` is useful when you need to send end-of-file to the child process, which causes the child process to exit, and then you want to obtain the exit status.

See also: `open-pipe`, `pipe-exit-status`

### `pipe-exit-status`  
**Function**

#### Summary
Returns the exit status of the child process that `open-pipe` created.

#### Package
`system`

#### Signature
`pipe-exit-status stream &key wait => exit-status, signal-number`

#### Arguments
- `stream`: A pipe stream.
- `wait`: A boolean.

#### Values
- `exit-status`: An integer, `nil` or the keyword `:closed`.
- `signal-number`: An integer or `nil`.

#### Description
The function `pipe-exit-status` returns the exit status of the child process that `open-pipe` created.

Stream must be a pipe stream object which was returned by a call to `open-pipe` with `save-exit-status` non-nil.

If `wait` is `t` then `pipe-exit-status` waits until the child process dies. If `wait` is `nil` then `pipe-exit-status` does not wait, and if the child process is still running it returns `nil`. The default value of `wait` is `t`.

On Microsoft Windows, if `close` was called on the stream before the child process died, then `pipe-exit-status` returns `exit-status :closed`. On Unix-like systems it works
after close but for compatibility it should be called only after calling close. If you need to send an end-of-file to the child process but also want to read the exit status, use pipe-close-connection before calling pipe-exit-status, and call close afterwards.

If exit-status is not nil or :closed, it is an integer which is the exit status of the child process. See “Interpreting the exit status” on page 462 for the interpretation of the exit status and the signal number.

See also open-pipe call-system pipe-kill-process pipe-close-connection “Interpreting the exit status” on page 462

pipe-kill-process

Function

Summary Tries to kill the process of a pipe stream.

Package system

Signature pipe-kill-process pipe-stream => result

Arguments pipe-stream A pipe stream.

Values result A boolean.

Description The function pipe-kill-process tries to kill the process of a pipe stream. pipe-stream must be the result of open-pipe. pipe-kill-process tries to kill the process that open-pipe creates.
The return value nil means that the process has already died. In this case the process is guaranteed to have died.

The return value t means that the process was still alive when pipe-kill-process was called, and it tried to kill it.

On Microsoft Windows, it causes the process to exit, but there may be some delay until it actually exits, so the process may still be alive at the time pipe-kill-process returns.

On Unix it sends SIGTERM to the process, which normally would cause it to exit, but in principle the process may handle SIGTERM and continue to run.

Notes
1. When open-pipe is called with a string, it executes it using a shell (non-Windows) or cmd (Windows), so the process that pipe-kill-process will kill is the shell or cmd. When open-pipe is called with a list, it executes the process (first element of the list) directly, and pipe-kill-process kills this process (the different behavior is actually documented in call-system).

2. On Microsoft Windows, pipe-kill-process needs to be called before the stream is closed by close. On Unix-like systems it works after close too, but for compatibility pipe-kill-process should not be called after close.

3. On Unix-like systems, if open-pipe was called with save-exit-status nil, there is a possibility that the child process that open-pipe started died and another process started with the same Process ID, and then pipe-kill-process may wrongly kill the new process. When open-pipe is called with save-exit-status non-nil, the status of the child process is tracked properly, and pipe-kill-process is guaranteed to do the right thing. On Windows pipe-kill-process always does the right thing.

4. When open-pipe was called with a string as the command, the process killed is the shell (Unix) or cmd (Windows), which normally kills the child process too. On
Unix-like systems the shell may execute the child process directly (overwriting itself with the child without forking) in which case it will kill the child. If you want to guarantee killing of the actual child, pass the command to `open-pipe` as a list of a vector.

See also
- `open-pipe`
- `call-system`

### pointer-from-address Function

**Summary**
Returns the object into which the given address is pointing.

**Package**
`system`

**Signature**
`pointer-from-address address => object`

**Arguments**
- `address` An integer giving the address of the object.

**Values**
- `object` The object pointed to by `address`.

**Description**
The function `pointer-from-address` returns the object into which the given integer `address` is pointing. Note that this address may not be pointing into this object after a garbage collection, unless the object is static and is still referenced by another Lisp variable or object.
Example

CL-USER 8 > (setq static-string
    (make-array 3
      :element-type 'base-char
      :allocation :static))
Warning: Setting unbound variable STATIC-STRING
"
"
CL-USER 9 > (sys:object-address static-string)
537166552
CL-USER 10 > (sys:pointer-from-address *)
"
"
CL-USER 11 > (eq * static-string)
T

See also

object-address
object-pointer

print-pretty-gesture-spec

Function

Summary
Prints a Gesture Spec object as a keystroke.

Package
system

Signature
print-pretty-gesture-spec gspec stream &key force-meta-to-alt
force-shift-for-upcase => gs泄

Arguments

gspec A Gesture Spec object.
stream An output stream.
force-meta-to-alt A boolean.
force-shift-for-upcase A boolean.

Values

gspec The Gesture Spec object that was passed.
**Description**  
The function `print-pretty-gesture-spec` prints the key-stroke represented by the Gesture Spec object `gspec` to the stream `stream`.

If `force-meta-to-alt` is true, then `gesture-spec-meta-bit` is represented as `Alt` in the output; otherwise it is represented as `Meta`. `force-meta-to-alt` defaults to `nil`.

If `force-shift-for-upcase` is true and `gspec` represents uppercase input such as `A`, then the `Shift` modifier is printed, indicating that `Shift` is pressed to obtain the `A` character. `force-shift-for-upcase` defaults to `t`.

If `gspec` has a wild modifiers or data component (that is, `gesture-spec-modifiers` and/or `gesture-spec-data` return `nil`) then `<Wild>` appears in the output.

**See also**  
gesture-spec-data  
gesture-spec-meta-bit  
gesture-spec-modifiers  
make-gesture-spec

**Variable**  

* `print-symbols-using-bars`  

**Summary**  
Controls how escaping is done when symbols are printed.

**Package**  
`system`

**Initial value**  
`nil`

**Description**  
The variable `*print-symbols-using-bars*` controls how escaping is done when symbols are printed.

When the value is true, printing symbols that must be escaped (for example, those with names containing the colon character `:`) is done using the bar character `|` instead of the backslash character `\` in cases when the readtable case and `*print-case*` are both `:upcase` or both `:downcase`. 
Example

CL-USER 1 > readtable-case *readtable* :UPCASE

CL-USER 2 > (let ((sys:*print-symbols-using-bars* t)
                   (*print-case* :upcase))
            (print (intern "FOO:BAR"))
            (values))

| FOO:BAR |

CL-USER 3 > (let ((sys:*print-symbols-using-bars* t)
                   (*print-case* :downcase))
            (print (intern "FOO:BAR"))
            (values))

foo:bar

product-registry-path

Function

Summary
Gets or sets a registry path for use with your software.

Package
system

Signature
product-registry-path product => path-string

Signature
(setf product-registry-path) path product => path

Arguments
product A Lisp object.

Values
path The path as a string or a list of strings.

path-string The path as a string.

Description
The function product-registry-path returns the registry subpath defined for the product denoted by product, as a string.

The function (setf product-registry-path) sets the registry subpath for the product denoted by product.
If *path* is a string it can contain backslash \ or forward slash / as directory separators - these are translated internally to the separator appropriate for the system. Note that any backslash will need escaping (with another backslash) if you input the string value via the Lisp reader.

If *path* is a list of strings, then it is interpreted like the directory component of a pathname.

This registry subpath is used when reading and storing user preferences with **user-preference**.

Note that while *product* can be any Lisp object, values of *product* are compared by **eq**, so you should use keywords.

**Notes**

To store CAPI window geometries under the registry path for your product, see the entry for **capi:top-level-interface-geometry-key** in the *CAPI User Guide and Reference Manual*.

**Example**

```lisp
(setf (sys:product-registry-path :deep-thought)
      (list "Deep Thought" "1.0"))
```

Then, on non-Windows systems:

```lisp
(sys:product-registry-path :deep-thought)
=>
"Deep Thought/1.0"
```

And on Microsoft Windows:

```lisp
(sys:product-registry-path :deep-thought)
=>
"Deep Thought\1.0"
```

**See also**

- **copy-preferences-from-older-version**
- **user-preference**
**release-object-and-nullify**

*Macro*

**Summary**
64-bit LispWorks only: Explicitly release the memory of an object if possible.

**Package**
`system`

**Signature**
```lisp
release-object-and-nullify place => released-p
```

**Arguments**

- `place` A generalized reference (see the Common Lisp definition).

**Values**

- `released-p` A boolean.

**Description**
The macro `release-object-and-nullify` checks if `place` contains a pointer to an object that can be explicitly released, and if it does then it frees the object and sets `place` to `nil`.

`place` must the only reference to the object to be released, and no other thread can be still accessing the object. `release-object-and-nullify` releases the memory and stores `nil` in `place`, so there is no dangling pointer to the object.

The released memory is returned to the operating system for objects that are allocated in their own segment. Currently, these are objects larger than 1 MB in the Mobile GC, larger than 64 MB in the ordinary 64-bit GC. For other objects, the memory will eventually be reclaimed by the GC, except for permanent objects that are not in their own segment, which are never reclaimed.

`release-object-and-nullify` cannot be executed in interpreted code (because the interpreted code would keep pointers to the object). It must be compiled.

**Notes**
If the pointer in `place` is not the only reference, then you will be left with "dangling" pointers to a free memory, with unpredictable results.
release-object-and-nullify is mainly intended to be used with references to the large objects that are allocated in their own segment, where it can return the memory to the operating system, and hence reduce the memory usage substantially without waiting for a GC. In particular, it is the only way to release the memory of such objects that were made permanent in the Mobile GC. This is the main purpose of it.

Since release-object-and-nullify does not release the memory for objects that are not in their own segment, it is not very useful for such objects. However, it may have a useful effect when called on an object that contains pointers and is in a higher generation, because objects in lower generations that are kept alive because of pointers from this object can be GCed earlier, but in most cases it probably does not justify the effort. release-object-and-nullify can be called with a reference to a permanent object too, and if it is allocated in its own segment then the memory will also be released.

release-object-and-nullify releases only the object that is referenced by place, but not anything that this object points to, which means that you may not get the effect you expect for an object that is complex, such as a hash-table, a CLOS instance, a pathname or a non-simple array.

See also

allocated-in-its-own-segment-p
make-object-permanent
make-permanent-simple-vector
“Mobile GC technical details” on page 154

---

room-values

Summary

Returns information about the state of internal memory.

Package

system
**Signature**

\[ \text{room-values} \Rightarrow \text{result} \]

**Values**

\[ \text{result} \]

A plist

\[ (:\text{total-size} \ \text{size} \ 
:\text{total-allocated} \ \text{allocated} \ 
:\text{total-free} \ \text{free}) \]

**Description**

`room-values` returns a plist containing information about the state of internal memory. This information is the same as would be printed by `(room nil)`.

**Notes**

In 64-bit LispWorks you can also use `count-gen-num-allocation` and `gen-num-segments-fragmentation-state`.

**See also**

`count-gen-num-allocation`
`room`

“Guidance for control of the memory management system” on page 130

---

**run-shell-command**

**Function**

**Package**

`system`

**Summary**

Allows executables and DOS or Unix shell commands to be called from Lisp code.

**Signature**

\[ \text{run-shell-command} \ \text{command} \ \&\text{key} \ \text{input} \ \text{output} \ \text{error-output} \ 
\text{separate-streams} \ \text{wait} \ \text{if-input-does-not-exist} \ \text{if-output-exists} \ \text{if-error-output-exists} \ 
\text{show-window} \ \text{environment} \ \text{element-type} \ \text{save-exit-status} \Rightarrow \text{result} \]

**Signature**

\[ \text{run-shell-command} \ \text{command} \ \&\text{key} \ \text{input} \ \text{output} \ \text{error-output} \ 
\text{separate-streams} \ \text{wait} \ \text{if-input-does-not-exist} \ \text{if-output-exists} \ \text{if-error-output-exists} \ 
\text{show-window} \ \text{environment} \ \text{element-type} \ \text{save-exit-status} \Rightarrow \text{stream, error-stream, process} \]
Arguments

command  A string, a list of strings, a simple-vector of strings, or nil.

input  nil, :stream or a file designator. Default value nil.

output  nil, :stream or a file designator. Default value nil.

error-output  nil, :stream, :output or a file designator. Default value nil.

separate-streams  A boolean. True value not currently supported.

wait  A boolean, default value t.

if-input-does-not-exist  :error, :create or nil. Default value :error.


show-window  A boolean. True value not currently supported.

environment  An alist of strings naming environment variables and values. Default value nil.

element-type  Default value base-char.

save-exit-status  A boolean, default value nil.

Values

result  The exit status of the process running command, or a process ID.

stream  A stream, or nil.

error-stream  A stream, or nil.
A process ID.

The function `run-shell-command` allows executables and DOS or Unix shell commands to be called from Lisp code with redirection of the stdout, stdin and stderr to Lisp streams. It creates a subprocess which executes the command `command`.

The argument `command` is interpreted as by `call-system`. In the cases where a shell is run, the shell to use is determined by the environment variable SHELL, or defaults to `/bin/csh` or `/bin/sh` if that does not exist.

If `wait` is true, then `run-shell-command` executes `command` and does not return until the process has exited. In this case none of `input`, `output` or `error-output` may have the value `:stream`, and the single value `result` is the exit status of the process that ran `command`.

If `wait` and `save-exit-status` are `nil` and none of `input`, `output` or `error-output` have the value `:stream` then `run-shell-command` executes `command` and returns a single value `result` which is the process ID of the process running `command`.

If `wait` is `nil` and either of `input` or `output` have the value `:stream` then `run-shell-command` executes `command` and returns three values: `stream` is a Lisp stream which acts as the stdout of the process if `output` is `:stream`, and is the stdin of the process if `input` is `:stream`. `error-stream` is determined by the argument `error-output` as described below. `process` is the process ID of the process.

If `wait` and `save-exit-status` are `nil` and neither of `input` or `output` have the value `:stream` then the first return value, `stream`, is `nil`.

If `wait` is `nil`, `save-exit-status` is true and neither of `input` or `output` have the value `:stream` then the first return value, `stream`, is a dummy stream that can only be used with `pipe-exit-status` (see `save-exit-status` below).
If \texttt{wait} is \texttt{nil} and \texttt{error-output} has the value \texttt{:stream} then \texttt{run-shell-command} executes \texttt{command} and returns three values. \texttt{stream} is determined by the arguments \texttt{input} and \texttt{output} as described above. \texttt{error-stream} is a Lisp stream which acts as the stderr of the process. \texttt{process} is the process ID of the process.

If \texttt{wait} is \texttt{nil} and \texttt{error-output} is not \texttt{:stream} then the second return value, \texttt{error-stream}, is \texttt{nil}. If \texttt{error-output} is \texttt{:output}, then stderr goes to the same place as stdout.

If \texttt{input} is a pathname or string, then \texttt{open} is called with \texttt{:if-does-not-exist if-input-does-not-exist}. The resulting \texttt{file-stream} acts as the stdin of the process.

If \texttt{output} is a pathname or string, then \texttt{open} is called with \texttt{:if-exists if-output-exists}. The resulting \texttt{file-stream} acts as the stdout of the process.

If \texttt{error-output} is a pathname or string, then \texttt{open} is called with \texttt{:if-exists if-error-output-exists}. The resulting \texttt{file-stream} acts as the stderr of the process.

This table describes the streams created, for each combination of stream arguments:

<table>
<thead>
<tr>
<th>Arguments</th>
<th>stream</th>
<th>error-stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{input} is \texttt{:stream} &lt;br&gt; \texttt{output} is \texttt{:stream} &lt;br&gt; error-output is \texttt{:stream}</td>
<td>An I/O stream connected to stdin and stdout</td>
<td>An input stream connected to stderr</td>
</tr>
<tr>
<td>\texttt{input} is not \texttt{:stream} &lt;br&gt; \texttt{output} is \texttt{:stream} &lt;br&gt; error-output is \texttt{:stream}</td>
<td>An input stream connected to stdout</td>
<td>An input stream connected to stderr</td>
</tr>
<tr>
<td>\texttt{input} is \texttt{:stream} &lt;br&gt; \texttt{output} is not \texttt{:stream} &lt;br&gt; error-output is \texttt{:stream}</td>
<td>An output stream connected to stdin</td>
<td>An input stream connected to stderr</td>
</tr>
</tbody>
</table>
Table 49.1  The streams created by `run-shell-command`

<table>
<thead>
<tr>
<th>Arguments</th>
<th>stream</th>
<th>error-stream</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>input</code> is not :stream <code>output</code> is not :stream <code>error-output</code> is :stream</td>
<td>nil</td>
<td>An input stream connected to stderr</td>
</tr>
<tr>
<td><code>input</code> is :stream <code>output</code> is :stream <code>error-output</code> is :output</td>
<td>An I/O stream connected to stdin, stdout and stderr</td>
<td>nil</td>
</tr>
<tr>
<td><code>input</code> is not :stream <code>output</code> is :stream <code>error-output</code> is :output</td>
<td>An input stream connected to stdout and stderr</td>
<td>nil</td>
</tr>
<tr>
<td><code>input</code> is :stream <code>output</code> is not :stream <code>error-output</code> is :output</td>
<td>An output stream connected to stdin</td>
<td>nil</td>
</tr>
<tr>
<td><code>input</code> is not :stream <code>output</code> is not :stream <code>error-output</code> is :output</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td><code>input</code> is :stream <code>output</code> is :stream <code>error-output</code> is not :stream or :output</td>
<td>An I/O stream connected to stdin and stdout</td>
<td>nil</td>
</tr>
<tr>
<td><code>input</code> is not :stream <code>output</code> is :stream <code>error-output</code> is not :stream or :output</td>
<td>An input stream connected to stdout</td>
<td>nil</td>
</tr>
<tr>
<td><code>input</code> is :stream <code>output</code> is not :stream <code>error-output</code> is not :stream or :output</td>
<td>An output stream connected to stdin</td>
<td>nil</td>
</tr>
<tr>
<td><code>input</code> is not :stream <code>output</code> is not :stream <code>error-output</code> is not :stream or :output</td>
<td>nil</td>
<td>nil</td>
</tr>
</tbody>
</table>
If any of input, output or error-output are streams, then they must be file-streams or socket-streams capable of acting as the stdin, stdout or stderr of the process.

environment should be an alist of strings naming environment variables and their values. The process runs in an environment inherited from the Lisp process, augmented by environment.

If save-exit-status is true then the system stores the exit status of the process, so that it can be recovered by calling pipe-exit-status on stream or error-stream if either of these is a stream.

Example

(multiple-value-bind (out err pid)
  (sys:run-shell-command "sh -c 'echo foo >&2; echo bar'"
    :wait nil
    :output :stream
    :error-output :stream)
  (with-open-stream (out out)
    (with-open-stream (err err)
      (values (read-line out) (read-line err))))))
=>
"bar", "foo"

See also
call-system
call-system-showing-output
open-pipe
pipe-exit-status

safe-locale-file-encoding

Function

Summary
Provides a safe encoding which corresponds to the current code page on Microsoft Windows, and the locale on Unix.

Package
system

Signature
safe-locale-file-encoding pathname ef-spec buffer length => new-ef-spec
Description
The function safe-locale-file-encoding is similar to locale-file-encoding except that it always returns a safe external format. That is, the external format does not signal error on writing characters not in the encoding.

On Microsoft Windows, safe-locale-file-encoding consults the ANSI code page. If the code page identifier \( id \) is in win32:*latin-1-code-pages*, it merges ef-spec with :latin-1-safe. This external format writes Latin-1 on output, using 63 (ASCII '?') to replace any non-Latin-1 characters that are written. If the code page identifier \( id \) is not in win32:*latin-1-code-pages* then safe-locale-file-encoding merges ef-spec with an encoding corresponding to the current code page that uses the code page’s replacement code for characters that cannot be encoded.

safe-locale-file-encoding merges ef-spec with :latin-1-safe on Unix.

See also
*file-encoding-detection-algorithm*
*latin-1-code-pages*
locale-file-encoding

---

**set-approaching-memory-limit-callback**

**Function**

**Summary**
Sets a callback that it is called when 32-bit LispWorks approaches its memory limit.

**Package**
*system*

**Signature**
set-approaching-memory-limit-callback *callback*

**Arguments**
callback A function designator.

**Description**
The function set-approaching-memory-limit-callback sets a callback that it is called when 32-bit LispWorks approaches its limit of memory.
The function `callback` must take two arguments: the size of the image and the margin of growth:

```lisp
(callback size margin)
```

 Normally `callback` should do something to prevent further growth of the image, or at least minimize the damage if LispWorks crashes when it actually reaches its limit (for example by saving data to disk).

`callback` can prevent an error being signalled by calling `cl:continue`.

If there is no callback (the default) or `callback` returns, LispWorks signals an error.

**Notes**

- `set-approaching-memory-limit-callback` is not relevant to 64-bit LispWorks.
- `set-approaching-memory-limit-callback` does not return a useful value.

**See also**

- "Approaching the memory limit" on page 142
- `approaching-memory-limit`

---

**set-automatic-gc-callback**

**Function**

**Summary**

Sets a function or functions to call after an automatic GC in 64-bit LispWorks.

**Package**

`system`

**Signature**

```lisp
(set-automatic-gc-callback blocking-gen-num-func &optional other-func => other-func)
```

**Arguments**

- `blocking-gen-num-func`  
  A function designator for a function of two arguments, or `nil`.  

other-func  A function designator for a function of one argument, or nil.

Values  other-func  A function designator for a function of one argument, or nil.

Description  The function set-automatic-gc-callback sets a function or functions to call after an automatic garbage collection (GC).

If blocking-gen-num-func is a function designator it should take two arguments: the generation number and, if do-gc in the last call to set-blocking-gen-num was a number, the number of copied segments. It is called whenever the blocking generation is garbage collected automatically. If blocking-gen-num-func is nil, then this callback is switched off.

If other-func is a function designator it should take one argument, the generation number that was garbage collected. It is called whenever an automatic GC occurred and blocking-gen-num-func was not called, either because the blocking generation was not garbage collected, or because blocking-gen-num-func was passed as nil. If other-func is nil (the default) then this callback is switched off.

The calls occur after the GC has finished and there is no restriction on what they can do. If the call ends up allocating enough to trigger another automatic GC, they enter again recursively.

Notes  set-automatic-gc-callback is implemented only in 64-bit LispWorks. It is not relevant to the Memory Management API in 32-bit implementations.

See also  set-blocking-gen-num
“Memory Management in 64-bit LispWorks” on page 146
**set-blocking-gen-num**

*Function*

**Summary**
Sets the blocking generation in 64-bit LispWorks.

**Package**
`system`

**Signature**
```
```

**Arguments**
- `gen-num` An integer between 0 and 7, inclusive.
- `do-gc` One of `t`, `nil` and `:mark`, or a real number between 0 and 10, inclusive.
- `max-size-to-copy` A positive real number, or `nil`.
- `max-size` A synonym for `max-size-to-copy`.
- `gc-threshold` An integer greater than 12800, or a real in the inclusive range `[0 100]`, or `nil`.

**Values**
- `old-blocking-gen-num` An integer between 0 and 7, inclusive.
- `do-gc` One of `t`, `nil` and `:mark`, or a real number between 0 and 10, inclusive.
- `max-size-to-copy` A positive real number.
- `old-gc-threshold` A number.

**Description**
The function `set-blocking-gen-num` sets `gen-num` as the generation that blocks. That is, no object is automatically promoted out of generation `gen-num` to a higher generation.

If `do-gc` is non-nil, then generation `gen-num` is automatically collected when needed, as defined by `gc-threshold` (see `set-gen-num-gc-threshold`).

The actual value of `do-gc` specifies how to GC the blocking generation when required. The possible values of `do-gc` are interpreted as follows:
t  Use Copying GC.
:mark Use Marking GC.

A number in the inclusive range [0, 10]

Use Marking GC with copying of fragmented segments. The value specifies the fragmentation-threshold (the same as the argument to marking-gc). This is the ratio between the amount of free space that cannot be easily used and the amount of allocated space inside a segment. Only segments with fragmentation higher than the threshold are copied.

The default value of do-gc is t.

max-size-to-copy is meaningful only if do-gc is a number. It specifies the maximum size in Gigabytes to try to copy. If the fragmented segments contain more data than this value, only some of them are copied in each GC.

If gc-threshold is non-nil, it is used to set the threshold for automatic GC using set-gen-num-gc-threshold.

The initial setup is as if this call has been made:

(sys:set-blocking-gen-num 3)

That is, the system will GC automatically according to the default gc-threshold using Copying GC.

Setting the blocking generation gen-num to a lower number is useful into two situations:

1. When you have an operation that allocates a significant amount of data, and almost of it goes when the operation finishes, it is useful to reduce the blocking gen-num during the operation. The macro block-promotion is a convenient way of doing that.
2. If you have a good idea of how your application behaves, it may be useful to block at a lower generation (2 or 1), and then periodically call \texttt{gc-generation} explicitly to promote long living objects to a higher generation. The advantage of doing this is that you can call \texttt{gc-generation} in places where you know there are not many short-lived objects alive.

Passing a \texttt{do-gc} value other than \texttt{t} is useful when the blocking generation can be large enough that copying it all may cause very serious paging. Passing \texttt{do-gc :mark} will stop the system from copying the blocking generation, but may cause fragmentation if a significant number of long-lived objects die after a while, and there are not explicit calls to \texttt{gc-generation} or \texttt{marking-gc}.

\texttt{set-blocking-gen-num} returns four values: the old blocking generation number, the old value of \texttt{do-gc}, the \texttt{max-size-to-copy}, and the old value of \texttt{gc-threshold}. It can be called with \texttt{gen-num nil} to query the values without changing any of them.

\textbf{Notes}

\texttt{set-blocking-gen-num} is implemented only in 64-bit LispWorks. It does nothing in the Mobile GC and its return value is not meaningful. It is not relevant to the Memory Management API in 32-bit implementations.

\textbf{See also}

\texttt{block-promotion}
\texttt{gc-generation}
\texttt{marking-gc}
\texttt{set-automatic-gc-callback}
\texttt{set-gen-num-gc-threshold}

“Guidance for control of the memory management system” on page 130
**set-default-segment-size**  

**Function**

**Summary**
Sets the default initial size of a segment in 64-bit LispWorks.

**Package**
`system`

**Signature**

```
set-default-segment-size gen-num allocation-type size-in-mb  
=> segment-size
```

**Arguments**

- `gen-num` An integer between 0 and 3, inclusive.
- `size-in-mb` A number, or `nil`.

**Values**

`segment-size` A number.

**Description**

The function `set-default-segment-size` sets the default initial size of a segment for a specific generation and allocation type.

The default initial size is also used as the default size for enlargement of the segment.

`allocation-type` can be any of the allocation types. However, if `allocation-type` is `:other-big` or `:non-pointer-big`, this function has no effect.

If `size-in-mb` is a number, it specifies the size in megabytes. If `size-in-mb` is `nil` then `set-default-segment-size` returns the default initial segment size without altering it.

The returned value, `segment-size`, is the previous default initial segment size.

During automatic garbage collections (GCs) the system collects an ephemeral generation when any of its segments for
the main allocation types is full. Thus the size of the segments defines the frequency of GCs in these generations.

**Notes**

`set-default-segment-size` is implemented only in 64-bit LispWorks. It does nothing in the Mobile GC and its return value is not meaningful. It is not relevant to the Memory Management API in 32-bit implementations, where `enlarge-generation` is available.

**See also**

`avoid-gc`

`enlarge-generation`

`set-maximum-segment-size`

“Memory Management in 64-bit LispWorks” on page 146

---

**set-delay-promotion**

*Function*

**Summary**

Delays promotion for a specified generation in 64-bit LispWorks.

**Package**

`system`

**Signature**

`set-delay-promotion gen-num on => on`

**Arguments**

`gen-num` An integer between 0 and 7, inclusive.

`on` A generalized boolean.

**Values**

`on` A generalized boolean.

**Description**

The function `set-delay-promotion` delays promotion for generation `gen-num`, which means that objects are promoted to the next generation in the second garbage collection (GC) that they survive in generation `gen-num`. By default, objects are promoted in the first GC.
It is not obvious under what circumstances delayed promotion is more useful than the default behavior. If you find this function useful, please let us know at Lisp Support.

Notes

set-delay-promotion is implemented only in 64-bit LispWorks. It does nothing in the Mobile GC and its return value is not meaningful. It is not relevant to the Memory Management API in 32-bit implementations.

See also

set-blocking-gen-num

set-expected-allocation-in-generation-2-after-gc

Function

Summary

Mobile GC only: tells the GC what is the maximum amount that you expect to be allocated in generation 2 after a GC of generation 2.

Package

system

Signature


Arguments

expected-other-mbs, expected-cons-mbs, max-needed-other-mbs, max-needed-cons-mbs

Non-negative integers or nil.

Values

prev-expected-other-mbs, prev-expected-cons-mbs, prev-max-needed-other-mbs, prev-max-needed-cons-mbs

Non-negative integers.

Description

The function set-expected-allocation-in-generation-2-after-gc is intended to improve the behavior of the application, but it may also degrade the performance if not used appropriately. It sets internal values associated with each of
the parameters `max-needed-other-mbs`, `max-needed-cons-mbs`, `expected-other-mbs` and `expected-cons-mbs`.

All parameters and return values are in megabytes. If any parameter is `nil` or is larger than the maximum, which is 65535, then the maximum is used for that parameter. `expected-other-mbs` defaults to its current internal value and `max-needed-other-mbs` defaults to the maximum of its current internal value and `expected-other-mbs` (see below). Likewise for `expected-cons-mbs` and `max-needed-cons-mbs`. LispWorks starts with all internal values set to the maximum.

The main purpose of `set-expected-allocation-in-generation-2-after-gc` is to tell the GC what you expect to be the maximum allocated megabytes in generation 2 after a GC of generation 2. That allows LispWorks to perform a better GC in situations where it does not have enough memory to copy all of generation 2. You set this separately for Other and `cons` objects, by supplying `expected-other-mbs` and `expected-cons-mbs`. Note that Other does not include Large and Static objects.

In situations where LispWorks cannot get enough memory from the operating system to copy all of generation 2, but can get the expected size that you have set, the GC is faster than it would have been if the expected size was not set (that is, set to the maximum), and more importantly, the memory usage after the GC will be smaller (sometimes much smaller). On the other hand, if the expected size is set too low in such a situation, then the GC is a little slower, and more importantly, the memory usage after the GC is larger than it would have been if the expected size was not set, until the next GC of generation 2 where the expectation is met. Thus for the expected size setting to be useful, it needs to be met (that is, the allocated size after GC must be less than the setting) in almost all GCs of generation 2. It probably needs to meet more than 90% in the GCs to be useful.
The function `get-maximum-allocated-in-generation-2-after-gc` is designed to allow you to find out what values to use. You exercise the application by trying to do anything that an end user may do, and then call `get-maximum-allocated-in-generation-2-after-gc` to see the maximums. Note that with normal settings, the points at which GC of generation 2 is invoked (and hence the amount alive after it) are not well defined. Therefore, you need to exercise the application more than once to find the correct numbers. Alternatively, if you block GC of generation 2 (by using `set-generation-2-gc-options`) and invoke the GC yourself, you can be more confident that you know the memory state at the time the GC is invoked. Alternatively, instead of calling `get-maximum-allocated-in-generation-2-after-gc` you can use the second and third return values of `(count-gen-num-allocation 2)`, or the values that `room` reports for "Other 2" and "Cons 2". You can use `set-automatic-gc-callback` to set a function that will be called immediately after GC of generation 2.

In situations where LispWorks can get all the memory it needs to perform a GC of generation 2, the setting of `expected-other-mbs` and `expected-cons-mbs` has no effect.

`max-needed-other-mbs` and `max-needed-cons-mbs` set an upper bound on the amount of memory that LispWorks tries to get from the operating system for a GC of generation 2.

`set-expected-allocation-in-generation-2-after-gc` ensures that the settings of `max-needed-other-mbs` and `max-needed-cons-mbs` are always equal or larger than the setting of `expected-other-mbs` and `expected-cons-mbs` respectively, by enlarging the setting of `max-needed-other-mbs` or `max-needed-cons-mbs` when needed.

Smaller values of `max-needed-other-mbs` and `max-needed-cons-mbs` cause LispWorks to use less memory in situations where it could get more memory from the operating system than they specify. That means the memory peak that happens dur-
ing the GC will be smaller. That should not have much effect the performance of LispWorks itself, but when the operating system is close to running out of memory, that may prevent it from actually running out of memory while the GC is running. On the other hand, if the GC requires more than max-needed-other-mbs or max-needed-cons-mbs, it will try to get more memory during the GC operation, and if this fails it has larger effect on performance than failure to allocate during the initialization of the GC.

The return values are the settings before the call.

set-expected-allocation-in-generation-2-after-gc

is thread-safe. It can be called repeatedly with different values.

Notes

set-expected-allocation-in-generation-2-after-gc

is useful in the situation when you have a "generation leak", that is objects live long enough to be promoted from generation 1 to 2 but die soon afterwards, and you do not have much allocated in generation 2 otherwise. In this case, the size of the live objects after a call to GC of generation 2 would be small, which means that the GC will be fast because the timing of a GC that works as planned is dependent on the amount allocated after the GC.

However, the memory peak will still be dependent on the size of generation 2 before the GC, which may cause failure to get memory from the operating system, and hence result in a slower and much less effective GC. Setting the values reduces the chance of such a failure, and reduces the memory usage even in situation where LispWorks could get the memory. Therefore, as long as the settings are correct (that is, in the vast majority of GCs of generation 2, the amount allocated after the GC is less than the setting) it can improve the performance of LispWorks significantly.

See also

get-maximum-allocated-in-generation-2-after-gc

count-gen-num-allocation
set-file-dates

Function

Summary
Sets the modification and access times of a file.

Package
system

Signature
set-file-dates file &key creation modification access

Arguments
file A pathname designator.
creation A non-negative integer, or nil.
modification A non-negative integer, or nil.
access A non-negative integer, or nil.

Description
The function set-file-dates sets the modification and access times of the file file for each of modification and access that is non-nil.

On Microsoft Windows, if creation is non-nil, the creation time of the file is also set. creation is ignored on other platforms.

Each keyword argument is interpreted as a universal time representing the time to set, unless it is nil in which case the corresponding time for file is not changed. Each keyword argument has default value nil.

An error of type file-error is signaled on failure.

See also
open
set-generation-2-gc-options

Function

Summary Mobile GC only: Controls the automatic GC of generation 2.

Package system

Signature set-generation-2-gc-options &key minimal-size-for-gc allocation-increase-factor => prev-minimal-size-for-gc, prev-allocation-increase-factor

Arguments minimal-size-for-gc nil, t or a non-negative fixnum.
allocation-increase-factor nil or a real between 0 and 16 (exclusive).

Values prev-minimal-size-for-gc Previous setting of minimal-size-for-gc.
prev-allocation-increase-factor Previous setting of allocation-increase-factor.

Description The function set-generation-2-gc-options sets internal variables that control when to automatically GC generation 2. For both arguments, the value nil (the default) causes no change to the corresponding internal variable.

minimal-size-for-gc is used to set the minimum allocated bytes in generation 2 that will trigger an automatic GC of generation 2. The special value t is interpreted as most-positive-fixnum, and effectively blocks any automatic GC of generation 2. Integers up to 10000 are interpreted as a size in megabytes, while integers above 10000 are interpreted as a size in bytes.

allocation-increase-factor controls how large an increase (since the last GC) in the allocated bytes in generation 2 will trigger another automatic GC of generation 2.

After each GC of generation 2 (automatic or user invoked), LispWorks computes a value for gen-2-gc-threshold using the expression:
\[
\text{(max \ minimal-size-for-gc}
\begin{align*}
&\quad (\star \ (1+ \ \text{allocation-increase-factor}) \\
&\quad \text{allocated-gen-2-bytes})
\end{align*}
\]

where \text{allocated-gen-2-bytes} is the allocated bytes in generation 2.

The next automatic GC of generation 2 will be triggered only when the allocated bytes in generation 2 exceeds \text{gen-2-gc-threshold} in the future.

\text{set-generation-2-gc-options} also sets \text{gen-2-gc-threshold} to \text{minimal-size-for-gc} if it is non-nil (\text{allocation-increase-factor} has no effect on this setting).

The initial values are 64 megabytes for \text{minimal-size-for-gc} and 0.5 for \text{allocation-increase-factor}.

Notes

Setting a small threshold causes relatively frequent GCs of generation 2, which is acceptable if each GC is short enough that it does not bother the end user. That may be true if the total allocated (after a GC) in your application is only few 10’s of megabytes. You should try to either time a GC of generation 2 by \text{(time (gc-generation 2))} on the slowest device to which you target your application, or alternatively try to use the application on such a device with the small threshold settings and see if you notice unacceptable delays.

A small threshold probably reduces the efficiency of the application a little, but has some advantages:

- It reduces the memory foot-print of your application.
- It reduces the chance that the application will run into a memory limitation when doing a GC. When the GC runs into such a limitation, it becomes much less efficient.
- It reduces the lifetime of objects in generation 0 and 1 that may stay alive because they are pointed to by objects in generation 2 that are actually dead but have not been GCed yet. As a result, in some circumstances frequent
GC of generation 2 may increase the efficiency of the application.

Therefore, if the delays are not long enough to be annoying then it is probably a good idea to have a low threshold.

By contrast, a large threshold causes less frequent GCs of generation 2, which normally causes it to grow more before a GC occurs. As long as LispWorks can get all the memory it needs to perform a copying GC, the time taken to do a GC correlates to the amount of memory that is still alive after (rather then before) the GC. Thus if generation 2 contains mainly "generation leak" objects (see “Preventing/reducing GC of generation 2” on page 159) that are removed by the GC, then each GC would take approximately the same time regardless of how frequently GC occurs. In that situation, doing a GC less frequently results in less time overall. However, if LispWorks fails to get memory from the operating system to do a copying GC, then the GC is done partially by a marking GC, which is slower and less effective. Thus you do not want to increase the threshold "too much". A copying GC needs to increase the memory size by approximately the current amount allocated, but it is difficult to predict how much memory the operating system is ready to give at any point, and it depends on the hardware and operating system. It is therefore difficult to say how much is "too much".

See also  
set-promote-generation-1
“Preventing/reducing GC of generation 2” on page 159

set-gen-num-gc-threshold

Function

Summary  
Sets the additional allocation threshold that triggers a GC in the blocking generation in 64-bit LispWorks.

Package  
system
Signature

\texttt{set-gen-num-gc-threshold\ gen-num\ threshold \Rightarrow old-threshold}

Arguments

\textit{gen-num} \hspace{1cm} An integer between 0 and 7, inclusive.

\textit{threshold} \hspace{1cm} An integer greater than 12800, or a real in the inclusive range \([0\ 100]\), or \texttt{nil}.

Values

\textit{old-threshold} \hspace{1cm} A number.

Description

The function \texttt{set-gen-num-gc-threshold} sets the threshold for additional allocation that triggers a garbage collection (GC) in generation \textit{gen-num} when this is the blocking generation (as set by \texttt{set-blocking-gen-num}). A GC is triggered when the allocation in generation \textit{gen-num} grows more than \textit{threshold} over the allocation after the last GC of this generation (or a GC of a higher generation).

To set the threshold, \textit{threshold} can be an integer greater than 12800, which is interpreted as the absolute value. Alternatively \textit{threshold} can be a real number in the inclusive range \([0\ 100]\), which is multiplied by the allocation since the previous GC to get the actual threshold to set.

The default threshold for all generations is 1. That is, for all generations \textit{gen-num}, when generation \textit{gen-num} is the blocking generation and allocation in it has doubled since the previous GC, generation \textit{gen-num} is collected automatically.

\texttt{set-gen-num-gc-threshold} can be called when the generation \textit{gen-num} is not the blocking generation, and will set the value for that \textit{gen-num}. Such a call will not take effect until the generation \textit{gen-num} becomes the blocking generation, as set by a call to \texttt{set-blocking-gen-num} (with \texttt{:do-gc\ non-nil}).

Increasing the threshold reduces the number of GC calls, but may increase the virtual memory usage.
set-gen-num-gc-threshold returns the old threshold for the generation gen-num. It can be called with threshold nil to return the threshold value without changing it.

Notes

set-gen-num-gc-threshold is implemented only in 64-bit LispWorks. It does nothing in the Mobile GC and its return value is not meaningful. It is not relevant to the Memory Management API in 32-bit implementations.

See also

set-blocking-gen-num

“Memory Management in 64-bit LispWorks” on page 146

set-maximum-memory

Function

Summary
Sets or removes a limit for the top of the Lisp heap in 32-bit LispWorks.

Package system

Signature set-maximum-memory address

Arguments address An integer address, or nil.

Description set-maximum-memory sets or removes a limit for the maximum address that the Lisp heap can grow to. If address is an integer, this becomes the maximum address. If address is nil, any limit set by set-maximum-memory is removed.

LispWorks sets the maximum memory on startup. In all cases the system is constrained by the size of the physical memory.

When the maximum memory is reached (either that set by set-maximum-memory or the physical memory limit) the system will become unstable. Therefore this situation should be avoided. The benefit of having the maximum memory set is that a useful error is signaled if the limit is reached.
An application which is likely to grow to the maximum memory should test the amount of available memory using `memory-growth-margin` or `room-values` at suitable times, and take action to reclaim memory. Do not rely on handling the error signaled when the maximum memory is reached, since the system is already unstable at this point.

Notes

`set-maximum-memory` is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations.

See also

- `check-fragmentation`
- `mark-and-sweep`
- `memory-growth-margin`
- `room-values`
- “Memory Management in 32-bit LispWorks” on page 135

---

**set-maximum-segment-size**

*Function*

**Summary**

Defines the maximum segment size for a generation and allocation type in 64-bit LispWorks.

**Package**

`system`

**Signature**

`set-maximum-segment-size gen-num allocation-type size-in-mb`

**Arguments**

- `gen-num` An integer between 0 and 7, inclusive.
- `allocation-type` One of :cons, :symbol, :function, :non-pointer and :other
- `size-in-mb` An integer between 1 and 256 inclusive, or nil.

**Values**

- `max-segment-size`
  A number.
**Description**

The function `set-maximum-segment-size` sets the maximum segment size for a generation and allocation type in 64-bit LispWorks.

`allocation-type` can be any of the main allocation types described in “Segments and Allocation Types” on page 147.

`size-in-mb` is the size in megabytes.

For the non-ephemeral generations (that is, the blocking generation and above), if the system needs more memory of some allocation type in some generation, its normal operation is to enlarge one of the existing segments in this generation of this allocation type. If it does not find a segment that it can enlarge, it allocates a new segment of the same allocation type in the same generation. Therefore the maximum segment size affects the number of segments that will be used.

There is an overhead to using more segments, so normally having the largest segment size which the implementation allows (256MB) is the best. Reducing the size may be useful when using `marking-gc` with `what-to-copy` non-nil or `set-blocking-gen-num` with `do-gc` a number to prevent fragmentation in the blocking generation. In this situation, reducing the size of each segment makes it easier for the system to find segments to copy, even if the `max-size-to-copy` parameter is set to a low number to avoid using too much virtual memory.

The returned value, `max-segment-size`, is the previous maximum segment size.

If `size-in-mb` is a number, it specifies the size in megabytes. If `size-in-mb` is `nil` then `set-maximum-segment-size` returns the maximum segment size without altering it.

**Notes**

`set-maximum-segment-size` is implemented only in 64-bit LispWorks. It does nothing in the Mobile GC and its return value is not meaningful. It is not relevant to the Memory Management API in 32-bit implementations.
See also

marking-gc
set-blocking-gen-num
set-default-segment-size

“Memory Management in 64-bit LispWorks” on page 146

set-memory-check

Function

Summary
Sets a memory check in 64-bit LispWorks.

Package
system

Signature
set-memory-check size function

Arguments
size An integer.
function A function designator.

Description
The function set-memory-check sets a memory check.
size must be an integer. It specifies the total size in bytes of
the mapped areas of Lisp at which the check is triggered.
function is a function of no arguments.

After each automatic garbage collection (GC) the system
checks whether the mapped area (excluding stacks) is larger
than size. If it is larger, function is called with no arguments.
Inside the dynamic scope of the call, the check is disabled.
There are no restrictions or special considerations on what
the function function does.

The current mapped area can be found by the :total-size
value returned by room-values.

Notes
set-memory-check is implemented only in 64-bit Lisp-
Works. It is not relevant to the Memory Management API in
32-bit implementations.
See also \texttt{set-memory-exhausted-callback}

\begin{description}
\item[set-memory-exhausted-callback \textit{Function}] \par
\begin{description}
\item[Summary] Sets a callback that is called when memory is exhausted in 64-bit LispWorks.
\item[Package] \textit{system}
\item[Signature] \texttt{set-memory-exhausted-callback function &optional where => callbacks}
\item[Arguments] \begin{description}
\item[function] A function designator, the keyword \texttt{:reset}, or \texttt{nil}.
\item[where] \texttt{:first}, \texttt{:last} or \texttt{nil}.
\end{description}
\item[Values] \texttt{callbacks} A list of function designators.
\item[Description] The function \texttt{set-memory-exhausted-callback} adds a callback that is called when memory is exhausted. That is, when the system fails to map memory.
\begin{description}
\item[Note:] \texttt{set-memory-check} is a more robust way to protect against memory exhaustion problems.
\item[If function is a function designator then it should be a function with signature] \texttt{function gen-num size type-name static}
\item[function is expected to report what the system was trying to allocate when it failed to map memory. Its arguments are:] \begin{description}
\item[gen-num] The number of the generation in which it was trying to allocate.
\item[size] The size in bytes which it was trying to allocate.
\end{description}
\end{description}
\end{description}
type-name A string naming the allocation type it was trying to allocate.

static A boolean, true if it was trying to allocate a static object, and false otherwise.

function can also have the special value :reset, which resets the callback list to nil.

function can also be nil, which means do nothing but simply return the current list of callbacks.

where defines the position in the list that the callback function is placed. Its allowed values are:

:first function is placed first in the callbacks list.
:last function is placed last in the callbacks list.
:nil function is removed from the callbacks list.

set-memory-exhausted-callback always first removes function from the callbacks list, and then adds it according to where. The default value of where is :first. Functions in the list are compared with equalp.

set-memory-exhausted-callback returns the callback list.

When a callback is called, Lisp already failed to map memory. This means that you must not rely on the callback to do real work. It should therefore attempt only a minimal amount of work such as clean-ups and generating debug information. It should not try to do real work.

After all the callbacks are called, the system signals an error of type storage-exhausted. The condition can be accessed using the accessors described for storage-exhausted.

Notes set-memory-exhausted-callback is implemented only in 64-bit LispWorks. It is not relevant to the Memory Management API in 32-bit implementations.
See also  
set-memory-check  
storage-exhausted

**set-promote-generation-1**

*Function*

**Summary**  
Mobile GC only: Set whether promotion occurs from generation 1 to 2.

**Package**  
system

**Signature**  
set-promote-generation-1 on &optional promote-all => prev-value

**Arguments**  
on  
A boolean.

promote-all  
nil, t or :full-gc.

**Values**  
prev-value  
A boolean.

**Description**  
The function **set-promote-generation-1** controls whether promotion can occur from generation 1 to 2. LispWorks starts with promotion on, so objects in generation 1 that survive a GC are promoted to generation 2. Calling **set-promote-generation-1** with on nil changes the behavior to leave those objects in generation 1.

**set-promote-generation-1** can be repeatedly called to switch promotion on or off. It affects all threads.

When on is nil and promote-all is non-nil then **set-promote-generation-1** promotes all currently live objects to generation 2, and then switches off promotion of generation 1. If promote-all is t, this promotion is done by a GC of generation 1. If promote-all is :full-gc, it is done by a GC of generation 2.

**set-promote-generation-1** returns the previous setting.

**Notes**  
Blocking promotion of generation 1 prevents "generation leaks", that is promotion of objects to generation 2 that die
not long afterwards, but it causes growth of generation 1, which makes GC of generation 1 slower. As long as generation 1 is not too large, then that is not a problem, and blocking the potential leaks into generation 2 is useful. If generation 1 grows as a result of blocking promotion, GC of generation 1 starts to take noticeable time, and it is better not to block but to tune generation 2.

See also “Preventing/reducing GC of generation 2” on page 159

**set-reserved-memory-policy**

*Function*

**Summary**

Mobile GC only: Tell LispWorks how much reserved memory to try to keep.

**Package**

system

**Signature**

`set-reserved-memory-policy fixed-size => policy`

**Arguments**

`fixed-size` A real or nil.

**Values**

`policy` nil or integer.

**Description**

The function `set-reserved-memory-policy` tells LispWorks how much reserved memory to keep.

If `fixed-size` is not `nil`, it tells LispWorks that it should keep this number of bytes of reserved memory. The reserved memory is held in segments of fixed size of 8 MB, the actual amount reserved is `fixed-size` rounded up to the nearest 8 MB.

If `fixed-size` is `nil`, LispWorks uses the total size of generation 0 plus generation 1 after each GC to compute the size to keep.

LispWorks starts with the setting being `nil`.

Calling `set-reserved-memory-policy` does not change the current size of the reserved area. LispWorks only reduces the reserved area after a GC if it is too large. It only increases it,
up to the limit, when the GC has copied all objects from a segment.

`set-reserved-memory-policy` returns the current setting.

See also “Mobile GC technical details” on page 154

**set-signal-handler**

**Function**

**Summary**
Installs or removes a handler for a POSIX signal.

**Note:** applicable only on non-Windows platforms.

**Package**
`system`

**Signature**
`set-signal-handler signum handler`

**Arguments**

- `signum` A POSIX signal number.
- `handler` A function or `nil`.

**Description**
`set-signal-handler` with a function `handler` configures LispWorks such that `handler` is called when the POSIX signal `signum` occurs.

If `handler` is `nil`, any handler for `signum` is removed.

`handler` should be defined to take an `&rest` argument, and ignore it. There are no restrictions on `handler` other than those applying to any asynchronous function call, and that it may be called in any thread. In particular there is no need to handle the signal immediately.

The configuration established by `set-signal-handler` is not persistent over image saving (or application delivery), so it should be called each time the image (or application) is started.
Notes

The currently defined signal handlers are shown in the output of the bug report template which can generated via the \texttt{bug-form} listener command. For example, there is a \texttt{SIGINT} handler which calls \texttt{break}. You should consult Lisp Support before overwriting existing signal handlers.

LispWorks initially has no \texttt{SIGHUP} handler. \texttt{SIGHUP} will kill a LispWorks process which does not have a \texttt{SIGHUP} handler installed. When the LispWorks IDE starts up, a \texttt{SIGHUP} handler (which attempts to release locks in the environment) is installed. However if you need a \texttt{SIGHUP} handler in a server application, for example, you should install one using \texttt{set-signal-handler}.

Example

\begin{verbatim}
(defun my-hup-handler (&rest x)
 (declare (ignorable x))
 (cerror "Continue"
 "Got a HUP signal"))

(sys:set-signal-handler 1 'my-hup-handler)
\end{verbatim}

Note that the LispWorks IDE overwrites a \texttt{SIGHUP} handler, so you would need to reinstall it after GUI startup.

\textbf{set-spare-keeping-policy}

\textit{Function}

Summary

Controls the behavior of the system when a segment is emptied in 64-bit LispWorks.

Package

\texttt{system}

Signature

\texttt{set-spare-keeping-policy gen-num policy => old-policy}

Arguments

\begin{itemize}
  \item \texttt{gen-num} An integer in the inclusive range \([0,7]\).
  \item \texttt{policy} A generalized boolean.
\end{itemize}

Values

\begin{itemize}
  \item \texttt{old-policy} A generalized boolean.
The function `set-spare-keeping-policy` controls the behavior of the system when a segment is emptied in 64-bit LispWorks.

If `policy` is non-nil, then when a segment in generation `gen-num` is emptied by copying all the objects out from it, it may be kept as a spare segment to be used in the future. This increases the use of virtual memory, but reduces the number of calls to `mmap` and `munmap`. It may be useful in applications that allocate at a very high rate.

If timing an application reveals a lot (more than 5%) of time in the “System Time”, and especially if this shows up in the GC times produced by `extended-time`, it may be useful to set the policy to non-nil in generation 1, 2 and maybe in generation 3.

The default policy is `nil` for all generations, meaning that empty segments are discarded.

The returned value `old-policy` is the previous policy for the generation `gen-num`.

`set-spare-keeping-policy` is implemented only in 64-bit LispWorks. It does nothing in the Mobile GC and its return value is not meaningful. It is not relevant to the Memory Management API in 32-bit implementations.

See also `extended-time`

---

### set-split-promotion

**Function**

Mobile GC only: Sets splitting promotion of generation 1.

**Summary**

**Package**

`system`

**Signature**

`set-split-promotion on-p => prev-on-p`
Arguments  

*on-p*  
A boolean.

Values  

*prev-on-p*  
A boolean.

Description  
The function `set-split-promotion` switches split promotion of generation 1 on or off, depending of the value of *on-p*. Split promotion means that in a copying (the default) GC, objects in generation 1 that already survived a GC of generation 1 are promoted to generation 2, while objects that are new in generation 1 stay in generation 1. Non-split promotion means that all objects in generation 1 are promoted to generation 2.

Split promotion makes it less likely that objects will reach generation 2 and then die (causing a "generation leak"), but means that the GC spends more time on long-lived objects in generation 1 that should be in generation 2. Since "generation leak" is the more serious problem, the default is on, and it is probably rarely useful to switch it off.

A situation when it is useful to switch it off is when you have an "initialization" phase when you allocate mostly long-lived objects, and explicitly invoke a GC of generation 2 at the end of this phase. In this situation, you are not worried about generation leak, because all the leaked objects will be discarded when you invoke the GC of generation 2, so switching off split promotion during the phase may speed it up. However the effect is unlikely to be large, and you should time the initialization phase with and without split promotion to see which is faster.

`set-split-promotion` returns the previous setting.

See also  
"Mobile GC technical details" on page 154
set-static-segment-size  

**Summary**  
Mobile GC only: set the default static segment size, and optionally the size of the initial segment.

**Package**  
system

**Signature**  
`set-static-segment-size size &optional init-size => prev-size, prev-init-size`

**Arguments**  
- `size`: A positive integer in the range #x10000 to #x1000000 or `nil`
- `init-size`: A positive integer in the range #x10000 to #x1000000 or `nil`

**Values**  
`prev-size, prev-init-size`  
Integer.

**Description**  
The function `set-static-segment-size` sets the size of new static segments other than the initial one, and optionally the size of the initial one.

Normally you should not use static objects, because it makes the GC less efficient. However, in some circumstances it may be the best solution. In this situation `set-static-segment-size` can be used to minimize the overhead.

The overhead for the GC depends on the number of segments, so it is best to minimize the number of static segments by making them larger. On the other hand, if a segment is not full, it wastes memory. Hence the ideal solution would be the smallest single segment that is large enough to accommodate all static allocation. You can check the current number and sizes of static segments by looking at the output of `(room t)`.

By default, LispWorks allocates a small segment on startup of size 64 kB (#x10000). It actually uses very little of it (a few hundred bytes, up to several kB if you have many processes), and if your application uses only a few kB that should be
enough. If there is further static allocation and the initial segment becomes full, then LispWorks allocates another segment of size 1 MB, and repeats this as needed.

`set-static-segment-size` controls these sizes. `size` controls the size of all the static segments except the one that is allocated on startup. `init-size` controls the size of the initial segment.

The initial segment is allocated before your code is called on startup, so to set the initial segment size you need to call `set-static-segment-size` before delivering the image. `set-static-segment-size` gives an error if `init-size` is non-nil and the Mobile GC is already running.

A nil value means do not change the value. Calling `set-static-segment-size` with nil is a way to get the current settings.

`set-static-segment-size` returns the previous settings.

See also “Mobile GC technical details” on page 154

**set-temp-directory**

*Function*

**Summary**
Sets the default temp directory.

**Package**
`system`

**Signature**
`set-temp-directory temp-dir`

**Arguments**
`temp-dir` A pathname or nil.

**Description**
The function `set-temp-directory` sets the default temp directory, that is the directory that `get-temp-directory` returns, and which is also used by `create-temp-file` and `open-temp-file`.

1875
temp-dir must be either a pathname of a suitable directory, or nil, which means use the default. The default is what the Operating System returns for a temp directory.

Notes  
set-temp-directory affects the global setting, that is all threads, and it is not thread-safe. If you need to call it, do that during start up. When you want to use temp files not in the default temp directory, you should call open-temp-file or create-temp-file with a suitable directory argument.

See also  
open-temp-file  
create-temp-file  
get-temp-directory

**setup-atomic-funcall**

**Function**

**Summary**  
Sets up mutually atomic funcalls in SMP LispWorks.

**Package**  
**system**

**Signature**  
setup-atomic-funcall &rest function-and-arguments

**Arguments**  
function-and-arguments A list.

**Description**  
The function **setup-atomic-funcall** sets up a funcall which will be executed atomically with respect to any other calls which were also set up by **setup-atomic-funcall**.

The call causes the execution of the form

```
(apply (car function-and-arguments)
       (cdr function-and-arguments))
```

some time after the entry to **setup-atomic-funcall**. The call may happen before **setup-atomic-funcall** returns, and it is expected that normally this is what will happen. However, it may be delayed for an indefinite period, but normally this period is short (milliseconds). The execution occurs
atomically with respect to other calls that were set up by `setup-atomic-funcall`.

The call should be short, because otherwise it will delay all the other calls. If an error occurs during the call, the atomicity is no longer guaranteed.

`setup-atomic-funcall` is useful when a process needs to atomically tell another process to do something, but does not need to wait for it to finish.

`setup-atomic-funcall` causes less congestion than using a lock, and so is more efficient for locks that may cause congestion. `compare-and-swap` and `atomic-exchange` operations will be faster.

See also

- `atomic-exchange`
- `compare-and-swap`

---

**`*sg-default-size*`**

*Variable*

**Summary**

Default initial size of a stack group.

**Package**

`system`

**Initial value**

- In LispWorks (64-bit) for Solaris:
  - `20000`
- In LispWorks (64-bit) on ARM64:
  - `18000`
- In all other implementations:
  - `16000`

**Description**

The value of the variable `*sg-default-size*` is the initial size of a stack group, in 32 bit words (in 32-bit implementations) or in 64 bit words (in 64-bit implementations).
*sg-default-size* can be bound around a call to a process creation function. Note that setting the global value of this variable affects the size of all system processes too, so this is not recommended.

Example
To create a process with a stack of 32000 words:

```lisp
(let ((sys:*sg-default-size* 32000))
  (mp:process-run-function "Larger stack" '()
    #'(lambda ()
        (print (hcl:current-stack-length)))))
```

See also
*current-stack-length*

*stack-overflow-behaviour*

description

The type *simple-int32-vector* provides simple vectors of int32 objects and can be used to generate optimal 32-bit arithmetic code. Create a *simple-int32-vector* by calling *make-simple-int32-vector*.

See the section “Fast 32-bit arithmetic” on page 472 for more information.

See also
int32
int32-aref
*make-simple-int32-vector*
*simple-int32-vector-length*
*simple-int32-vector-p*
simple-int32-vector-length

Function

Summary
Return the length of a simple-int32-vector.

Package
system

Signature
simple-int32-vector-length vector => length

Arguments
vector A simple-int32-vector.

Values
length A non-negative fixnum.

Description
The function simple-int32-vector-length returns the length of vector.

See also
make-simple-int32-vector
simple-int32-vector

simple-int32-vector-p

Function

Summary
A predicate for objects of type simple-int32-vector.

Package
system

Signature
simple-int32-vector-p object => result

Arguments
object An object.

Values
result A boolean.

Description
The function simple-int32-vector-p returns true if object is a simple-int32-vector and false otherwise.

See also
make-simple-int32-vector
simple-int32-vector
simple-int64-vector

**Type**

**Summary**
A type for simple vectors of int64 objects.

**Package**
system

**Signature**
simple-int64-vector

**Description**
The type `simple-int64-vector` provides simple vectors of int64 objects and can be used to generate optimal 64-bit arithmetic code. Create a `simple-int64-vector` by calling `make-simple-int64-vector`.

See the section “Fast 64-bit arithmetic” on page 474 for more information.

**See also**
int64
int64-aref
make-simple-int64-vector
simple-int64-vector-length
simple-int64-vector-p

---

simple-int64-vector-length

**Function**

**Summary**
Return the length of a `simple-int64-vector`.

**Package**
system

**Signature**
```
simple-int64-vector-length  vector  =>  length
```

**Arguments**

**vector**
A `simple-int64-vector`.

**Values**

**length**
A non-negative fixnum.

**Description**
The function `simple-int64-vector-length` returns the length of `vector`. 
See also
make-simple-int64-vector
simple-int64-vector

simple-int64-vector-p

Function
Summary A predicate for objects of type simple-int64-vector.
Package system
Signature simple-int64-vector-p object => result
Arguments object An object.
Values result A boolean.
Description The function simple-int64-vector-p returns true if object is a simple-int64-vector and false otherwise.
See also make-simple-int64-vector
simple-int64-vector

*specific-valid-file-encodings*

Variable
Summary List of external formats to check for validity.
Package system
Initial Value nil
Description The variable *specific-valid-file-encodings* is a list used by specific-valid-file-encoding, which tests each element of *specific-valid-file-encodings* to see if it is valid for the contents of buffer, bounded by length, and returns the first valid encoding.
See also  
*specific-valid-file-encoding*  
*file-encoding-detection-algorithm*

**specific-valid-file-encoding**  
*Function*

**Summary**  Provides an encoding chosen from list of specific encodings if valid.

**Package**  system

**Signature**  
`specific-valid-file-encoding pathname ef-spec buffer length`  
`=> new-ef-spec`

**Arguments**  
- `pathname`  Pathname identifying location of buffer.  
- `ef-spec`  An external format spec.  
- `buffer`  A buffer whose contents are examined.  
- `length`  Length (an integer) up to which buffer should be examined.

**Values**  
- `new-ef-spec`  Default external format spec created by merging ef-spec with the encoding that was found to be valid.

**Description**  The function `specific-valid-file-encoding` tests each element of *specific-valid-file-encodings* to see if it is valid for the contents of `buffer`, bounded by `length`. The first valid encoding is returned. For input files, `buffer` will contain the start of the file, so it is assumed that it contains a representative sample. For output files, `buffer` will have length 0, so the first element of *specific-valid-file-encodings* will always be returned.  
`pathname` is ignored.

**Notes**  You need to set *specific-valid-file-encodings* for `specific-valid-file-encoding` to have any effect. The
default value of *specific-valid-file-encodings* is nil, which causes specific-valid-file-encoding to return ef-spc unchanged.

See also  
*specific-valid-file-encodings*  
*file-encoding-detection-algorithm*

*stack-overflow-behaviour*  

**Variable**

Summary
Controls behavior when stack overflow is detected.

Package
system

Initial value
:error

Description
The variable *stack-overflow-behaviour* controls behavior when stack overflow is detected.

When *stack-overflow-behaviour* is set to :error, LispWorks signals an error.

When it is set to :warn, LispWorks increases the stack size automatically to accommodate the overflow, but prints a warning message to signal that this has happened.

When it is set to nil, LispWorks increases stack size silently.

Notes
Stack overflow is only detected when code was compiled with optimize qualities safety >= 1 or interruptible > 0 (see “Compiler control” on page 104l). Code compiled with safety = 0 and interruptible = 0 can cause an undetected stack overflow that will crash LispWorks.

Compatibility notes
In LispWorks 4.4 and previous on Windows and Linux platforms, automatic stack extension is not implemented. This has been fixed in LispWorks 5.0 and later.

See also
*sg-default-size*
The SYSTEM Package

**staticp**

*Function*

**Summary**

Specifies whether a given object has been allocated in static memory.

**Package**

`system`

**Signature**

`staticp obj => bool`

**Arguments**

`obj` An object.

**Values**

`bool` `t` if the object is allocated in static memory; `nil` otherwise.

**Description**

This predicate can be used on an object to find out whether it is allocated in static memory.

Foreign objects made by Lisp — for example in a Foreign Language Interface program — are made in static memory. The Lisp representations of these alien objects are not, however. Therefore `staticp` applied to an alien returns `nil` even though the alien instance itself is really allocated in static memory. To establish this, you can check the pointer to the alien instance within its Lisp representation (a structure).

**storage-exhausted**

*Class*

**Summary**

A condition class for failures to map memory.

**Superclasses**

`storage-condition`

**Initargs**

`:gen-num` The number of the generation in which the system was trying to allocate.

`:size` The size in bytes which the system was trying to allocate.
A string naming the allocation type the system was trying to allocate.

A boolean, true if the system was trying to allocate a static object, and false otherwise.

**Accessors**
- `storage-exhausted-gen-num`
- `storage-exhausted-size`
- `storage-exhausted-static`
- `storage-exhausted-type`

**Description**
The class `storage-exhausted` is a condition class used for reporting failures to map memory.

Allocation types are as described in `set-maximum-segment-size`.

**See also**
- `set-memory-exhausted-callback`

### `sweep-gen-num-objects`

**Function**

**Summary**
Applies a function to all the live objects in a generation in 64-bit LispWorks.

**Package**
`system`

**Signature**
`sweep-gen-num-objects gen-num function`

**Arguments**
- `gen-num` An integer in the inclusive range [0,7].
- `function` A designator for a function of one argument, the object.

**Values**
`sweep-gen-num-objects` returns `nil`.

**Description**
The function `sweep-gen-num-objects` applies `function` to all the live objects in the generation `gen-num`. 
function should take one argument, the object. It can allocate, but if it allocates heavily the sweeping becomes unreliable. Small amounts of allocation will normally happen only in generation 0, and so will not affect sweeping of other generations.

Notes  

sweep-gen-num-objects is not implemented in 32-bit LispWorks, where you can use sweep-all-objects instead.

sweep-gen-num-objects does not sweep cons objects in the Mobile GC.

See also sweep-all-objects

typed-aref  

Function

Summary  

Accesses a typed aref vector efficiently.

Package system

Signature typed-aref type vector byte-index => value

(setf typed-aref) value type vector byte-index => value

Arguments  

type  

A type specifier.

vector  

A vector created by make-typed-aref-vector.

byte-index  

A non-negative fixnum.

Values  

value  

An object of type type.

Description  

The function typed-aref allows efficient access to a typed aref vector.

The following values of type are accepted:

- double-float
• float
• single-float
• sys:int32
• (unsigned-byte 32)
• (signed-byte 32)
• (unsigned-byte 16)
• (signed-byte 16)
• (unsigned-byte 8)
• (signed-byte 8)
Additionally in 64-bit LispWorks only, the following values of type are also accepted:
• sys:int64
• (unsigned-byte 64)
• (signed-byte 64)

vector must be an object returned by make-typed-aref-vector.

byte-index specifies the index in 8-bit bytes from the start of the data in the vector. It must be a non-negative fixnum which is less than the byte-length argument passed to make-typed-aref-vector.

typed-aref and (setf typed-aref) will be inlined to code which is as efficient as possible when compiled with (optimize (safety 0)) and a constant type. As usual, you need to add (optimize (float 0)) to remove boxing for the float types.

Notes
Efficient access to foreign arrays is also available. See fli:foreign-typed-aref in the LispWorks Foreign Language Interface User Guide and Reference Manual
Example

(defun double-float-typed-aref-incf (x y z)
  (declare (optimize (float 0) (safety 0)))
  (incf (sys:typed-aref 'double-float x y)
    (the double-float z)))
  x)

See also make-typed-aref-vector
“Optimized integer arithmetic and integer vector access” on page 472

wait-for-input-streams

Function

Summary
Waits for input on a list of socket streams, returning those that are ready.

Package system

Signature
wait-for-input-streams streams &key wait-function wait-reason timeout => result

Arguments
streams A list, each member of which is a socket-stream.
wait-function A function of no arguments.
wait-reason A string.
timeout A real number or nil.

Values result A list of socket-streams or nil.

Description The function wait-for-input-streams waits for any of the streams in the argument streams to be ready for input. "Ready for input" typically means that some input is available from the stream, but can also mean that the peer closed the connection or there is an attempt to connect to the socket. Note that this function first checks the buffer for buffered streams.
When any of the streams is ready for input, \texttt{wait-for-input-streams} returns a list of all the streams that are ready, in the same order that they appear in \texttt{streams}.

If \texttt{timeout} is non-nil it must be a real number, specifying a timeout in seconds. If \texttt{timeout} seconds pass and none of the streams is ready, \texttt{wait-for-input-streams} returns \texttt{nil}.

If \texttt{timeout} is 0, \texttt{wait-for-input-streams} returns all of the streams that are ready immediately, without waiting at all. That is, it behaves like \texttt{listen} on many streams.

If \texttt{wait-function} is supplied, it is called periodically with no arguments, and if it returns non-nil then \texttt{wait-for-input-streams} returns \texttt{nil}. Note that, like the \texttt{wait-function} of \texttt{process-wait}, \texttt{wait-function} is called often and on other threads, so need to be an inexpensive call and independent of dynamic context.

If \texttt{wait-reason} is supplied it is used as the \texttt{wait-reason} for the Lisp process that calls \texttt{wait-for-input-streams} while it is waiting.

\textbf{Notes} \texttt{wait-for-input-streams} may return the list \texttt{streams} that was passed to it as is, if all the streams are ready.

\textbf{See also} \texttt{wait-for-input-streams-returning-first}

\textbf{wait-for-input-streams-returning-first} \quad \textit{Function}

\textbf{Summary} Waits for input on a list of socket streams, returning the first stream that is ready.

\textbf{Package} \texttt{system}

\textbf{Signature} \texttt{wait-for-input-streams-returning-first streams &key wait-function wait-reason timeout => result}
Arguments

- **streams**: A list, each member of which is a `socket-stream`.
- **wait-function**: A function of no arguments.
- **wait-reason**: A string.
- **timeout**: A real number or `nil`.

Values

- **result**: A `socket-stream` or `nil`.

Description

The function `wait-for-input-streams-returning-first` behaves just like `wait-for-input-streams` except that it returns the first stream in the list `streams` that is ready for input.

See also

`wait-for-input-streams`

---

**with-modification-change**

*Macro*

Summary

Provides a way to check whether there was any "modification" during execution of a body of code.

Package

`system`

Signature

`with-modification-change modification-place &body body`

Arguments

- **modification-place**: A place as defined in Common Lisp which can receive a fixnum.
- **body**: Lisp code

Description

The macro `with-modification-change`, together with the macro `with-modification-check-macro`, provides a way for a body of code to execute and check whether there was any "modification" during this execution, where modification is execution of some other piece of code.
See “Aids for implementing modification checks” on page 300 for the full description and an example.

Notes modification-place does not need to be one of the places defined for low level atomic operations.

See also with-modification-check-macro

with-modification-check-macro

Macro

Summary Provides a way to check whether there was any "modification" during execution of a body of code.

Package system

Signature with-modification-check-macro macro-name modification-place &body body

Arguments modification-place A place as defined in Common Lisp which can receive a fixnum.

Description The macro with-modification-check-macro, together with the macro with-modification-change, provides a way for a body of code to execute and check whether there was any "modification" during this execution, where modification is execution of some other piece of code.

with-modification-check-macro defines a lexical macro (by macrolet) with the name macro-name which takes no arguments, and is used to check whether there was any change since entering the body.

modification-place must be initialized to a fixnum. It must not be modified by any code except with-modification-change.

See “Aids for implementing modification checks” on page 300 for the full description and an example.
Notes

modification-place does not need to be one of the places defined for low level atomic operations.

See also

with-modification-change

with-other-threads-disabled

Macro

Summary

A debugging macro which executes code with all other threads temporarily disabled.

Package

system

Signature

with-other-threads-disabled &body body => results

Arguments

body Code.

Values

results The results of evaluating body.

Description

The macro with-other-threads-disabled disables all the other threads (that is, the mp:process objects), executes body and then enables the other threads. Thus it guarantees "single-thread execution" for the forms in body.

The point at which each of the other threads is stopped is not well-defined. It is always a GC safe point, but it can be inside manipulating some data structure or while holding a lock. As a result, if the code in body accesses a data structure or tries to lock a lock, it may see an inconsistent structure or get an error about calling process-wait when scheduling not is allowed.

As a result, with-other-threads-disabled is safe only if the code in body does not do anything that accesses trees of pointers and expects them to be in a consistent state and does not use locks. Any other code may, rarely but not never, get some unexpected error.

with-other-threads-disabled is useful for:
• the most accurate timing possible of specific operations
• running `sweep-all-objects` reliably
• "freezing" the program when something unexpected occurs and you want to debug it in the terminal.

Notes

`with-other-threads-disabled` cannot be guaranteed to be 100% safe in all cases, and therefore must not be used in end-user applications. It is useful for debugging purposes.

The LispWorks IDE relies on multithreading and will not work while the code in `body` executes.

See also

`sweep-all-objects`
`time`
The SYSTEM Package
This chapter describes miscellaneous symbols available in the WIN32 package. The WIN32 package also includes “The Windows registry API”, “The DDE client interface” and “The DDE server interface”. These are documented in separate chapters in this manual.

Note: the WIN32 package is not a supported implementation of the Win32 API. You should not use symbols in the WIN32 package unless they are documented in this manual. Instead, define your own interfaces to Windows functions as you need - see the LispWorks Foreign Language Interface User Guide and Reference Manual for details.

Note: This chapter applies only to LispWorks for Windows, and not the UNIX, Linux, x86/x64 Solaris, FreeBSD, AIX or Mac OS X platforms.

### canonicalize-sid-string

#### Function

**Summary**

Returns the canonical format of a SID specifier string.

**Package**

win32

**Signature**

`canonicalize-sid-string sid-string => result`
Arguments  

sid-string  A string.

Values  

result  A string or nil.

Description  
The function canonicalize-sid-string returns the canonical format of the SID specified by sid-string. If the string is already canonical (in the S-1-.. format) it returns a string which is equal to it. When the string is an alias, it returns the canonical form. The aliases are documented in the MSDN in the page titled "SID strings" (search for SDDL_ANONYMOUS).

canonicalize-sid-string returns nil for an unrecognized SID.

See also  
wait-for-connection
security-description-string-for-open-named-pipe

\textbf{connect-to-named-pipe}  

\textit{Function}

Summary  
Opens a stream connection to a named pipe.

Package  
\texttt{win32}

Signature  
connect-to-named-pipe name &key host errorp direction

Arguments  

name  A string.

host  A string or nil.

errorp  A boolean.

direction  One of the keywords :io, :input and :output.

Values  

stream  A stream or nil.

keyword  A keyword or nil.

condition  An error condition or nil.
Description

The function **connect-to-named-pipe** opens a connection to a named pipe and returns a stream connected to it that can be used like any other stream.

`name` is the pipe name. It can contain any character except `\` (according to the MSDN). For successful connection another process must have already created a pipe with that name, with the right input/output direction and permissions for the caller of **connect-to-named-pipe**, and tried to connect to it but has not succeeded yet. In LispWorks this is done by **open-named-pipe-stream**. The Windows function is `ConnectNamedPipe`.

`host`, if non-nil, specifies a host on which the named pipe was created. `host nil` means the current machine.

`direction` specifies the direction of input/output. It needs be allowed by the pipe (in inverse, that is if **connect-to-named-pipe** gets direction :input then the pipe must have been opened for output, for example by calling **open-named-pipe-stream** with `direction :output` or :io). The default value of `direction` is :io.

`errorp` specifies what to do in case of failure. If it is non-nil (the default), an error is signaled. If it is nil, then **connect-to-named-pipe** returns `stream nil`, `keyword` is a descriptive keyword, and `condition` is an error condition. `keyword` can be one of:

- **:does-not-exist**  
  There is no named pipe with this name.

- **:all-in-use**  
  There is at least one named pipe with this name, but all are already connected.

- **:access-denied**  
  There is already a named pipe with this name, but it denies access. That may be either because the permissions of the named
pipe do not allow the connection, or because other security features of the host system block the connection.

:error An unknown error.

On success connect-to-named-pipe returns a stream and the other two returned values are both nil.

See also open-named-pipe-stream

dismiss-splash-screen  

Function

Summary Makes a startup screen disappear.

Package win32

Signature dismiss-splash-screen &optional forcep

Arguments forcep A generalized boolean.

Description The function dismiss-splash-screen makes a startup screen (as specified via the :startup-bitmap-file delivery keyword) disappear.

If forcep is nil then the startup screen is displayed for a minimum of 5 seconds before disappearing. If forcep is true then the startup screen disappears when dismiss-splash-screen is called. The default value of forcep is nil.

If dismiss-splash-screen is not called, the startup screen appears for 30 seconds.

Note: the user can dismiss the startup screen by clicking on it.

For more information about specifying a startup screen in your application, see the entry for :startup-bitmap-file in the LispWorks Delivery User Guide.
**impersonating-named-pipe-client**

**Macro**

**Summary**
Executes code while impersonating the client of the named pipe.

**Package**
win32

**Signature**

```lisp
impersonating-named-pipe-client (named-pipe-stream &key fail-form fail-no-read-form) &body body
```

**Arguments**

- **named-pipe-stream** A named pipe stream.
- **fail-form** A Lisp form.
- **fail-no-read-form** A Lisp form.
- **body** Lisp forms.

**Description**
The macro `impersonating-named-pipe-client` executes the code of `body` while impersonating the client of the named pipe.

`named-pipe-stream` must be the result of `open-named-pipe-stream`.

For the impersonation to work, some input must have already been read from the pipe. If impersonation is used on a named pipe from which nothing was read, it calls `error` unless `fail-no-read-form` is supplied, in which case it executes this form. For all other kinds of failure `fail-form` is executed.

Apart from mechanism used to find the user to impersonate, `impersonating-named-pipe-client` behaves identically to `impersonating-user`. See `impersonating-user` for further details.

**Notes**
The limitation that some input must have been read is an undocumented restriction in the underlying Microsoft Windows functions.

**See also**

`impersonating-user`
impersonating-user  

**Macro**  

**Summary**  
Executes code while impersonating the user.

**Package**  
win32

**Signature**  
`impersonating-user (user-name password &key domain logon-type fail-form) &body body`

**Arguments**

- **user-name**  
  A string, or t.

- **password**  
  A string.

- **domain**  
  A string or nil.

- **logon-type**  
  nil or one of the keywords :interactive, :batch, :network, :network-cleartext, :service and :new-credentials.

- **fail-form**  
  A Lisp form.

- **body**  
  Lisp forms.

**Description**  
The macro `impersonating-user` executes the code of `body` while impersonating a specified user.

`user-name` and `password` specify login credentials. In general, these are strings but the symbol t as `user-name` is treated specially to mean the user that is currently interacting with the console of the computer (password is ignored in this case).

`domain`, if non-nil, must be a string specifying the domain or server where the account database to find the user is held. It can be "." meaning the local database. `domain nil` means use the default domain or server, as defined by Windows.

The `logon-type` keywords are mapped to the `LOGON32_LOGON_*` constants which are documented in the MSDN entry for `LogonUser`. The default value `nil` of `logon-type` is treated as an alias for :interactive.
This chapter applies only to LispWorks for Windows

body is evaluated only if the impersonation is successful. If the impersonation is not successful for any reason, body is not executed, and instead fail-form is evaluated.

On success, impersonating-user returns the result of the last form of body. On failure, it returns the result of fail-form.

Notes
Impersonation means that, in operations where the user identity makes a difference, the user identity is the impersonated user rather than the user running the process. For example, when opening a file the system uses the credentials of the impersonated user to check the access control list of the file. When creating a file, the impersonated user is also used as the owner and creator of the file.

The process that tries to impersonate must have special privilege to do that. Processes do not normally have these privileges. The processes that do are those that run with system credentials, for example services. Impersonation is used by these processes to perform specific operations with the credentials of some user rather than the system user.

Impersonation can also be used when a service process wants to start a process to interact with the user. In that situation, the new process must run as the user. To do that, you start process inside the scope of impersonating-user, for example by calling call-system or open-pipe. Normally you will want to run as the user that is currently logged on the console (see the special user-name value t above).

See also impersonating-named-pipe-client

known-sid-integer-to-sid-string

Function

Summary Returns the sid string for a known SID type integer

Package win32
Signature

\[
\text{known-sid-integer-to-sid-string } \text{integer} \Rightarrow \text{sid-string}
\]

Arguments

\text{integer} \quad \text{An integer.}

Values

\text{sid-string} \quad \text{A string or nil.}

Description

The function \text{known-sid-integer-to-sid-string} returns the SID string for \text{integer}, which needs to be one of the known integers, which are documented in the MSDN in the entry for \text{WELL_KNOWN_SID_TYPE Enumeration}.

\text{known-sid-integer-to-sid-string} \text{ returns nil for an unknown integer.}

See also

\text{wait-for-connection}
\text{security-description-string-for-open-named-pipe}

*latin-1-code-pages*

Variable

Summary

Windows Code Pages for which Latin-1 encoded files are used.

Package

\text{win32}

Initial value

\( (1252 \ 28591) \)

Description

The value of \text{*latin-1-code-pages*} is a list of integers, which must be Windows code page identifiers. When the current Code Page is on this list, the default file encoding detection algorithm will cause \text{(\text{:latin-1 :encoding-error-action 63})} to be used for file I/O. Files will be written as Latin-1 with '?' replacing any non-Latin-1 character. This is faster than converting to the code page.
If `safe-locale-file-encoding` is used for file encoding detection, then the `:latin-1-safe` external format will be used.

**Notes**

The LispWorks editor binds `*latin-1-code-pages*` to `nil` when reading and writing files, in order to ensure that code page characters outside of Latin-1 are handled regardless of the configuration of `open`.

**See also**

`*file-encoding-detection-algorithm*`

---

### `long-namestring`

**Summary**

Returns the long form of a namestring.

**Package**

`win32`

**Signature**

`long-namestring pathname => result`

**Arguments**

- `pathname` A pathname designator.

**Values**

- `result` A string or `nil`.

**Description**

The function `long-namestring` first obtains the full namestring as if by `cl:namestring`, and then converts this namestring to the long form (in the Microsoft Windows meaning of "Long" paths).

If the translation succeeds then `result` is a string in the Long form.

The translation may fail, in which case `nil` is returned.

**See also**

`short-namestring`
*multibyte-code-page-ef*  

**Variable**  

<table>
<thead>
<tr>
<th>Summary</th>
<th>Holds the external format corresponding to the current Windows multi-byte code page.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>win32</td>
</tr>
<tr>
<td>Description</td>
<td>This variable holds the external format corresponding to the current Windows multi-byte code page. It is automatically initialized to the right value, when the image is started. If you change the code page (using _setmbcp), you need to set this variable, too.</td>
</tr>
<tr>
<td>See also</td>
<td>locale-file-encoding</td>
</tr>
</tbody>
</table>

**named-pipe-stream-name**  

**Function**  

<table>
<thead>
<tr>
<th>Summary</th>
<th>Returns the name of a named pipe stream.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>win32</td>
</tr>
<tr>
<td>Signature</td>
<td>named-pipe-stream-name stream =&gt; name</td>
</tr>
<tr>
<td>Arguments</td>
<td>stream</td>
</tr>
<tr>
<td>Values</td>
<td>name</td>
</tr>
<tr>
<td>Description</td>
<td>The function named-pipe-stream-name returns the name of a named pipe stream. stream must the result of a call to open-named-pipe-stream. name is the name of the stream, that is, the argument to open-named-pipe-stream.</td>
</tr>
<tr>
<td>See also</td>
<td>wait-for-connection</td>
</tr>
</tbody>
</table>
open-named-pipe-stream

**Function**

**Summary**
Creates a stream that writes and read through a named pipe.

**Package**
win32

**Signature**
open-named-pipe-stream name &key errorp allow-remote max-number wait-reason timeout wait-function direction inherit-access-p access => stream, connectedp, condition

**Arguments**
- **name**: A string identifying the pipe.
- **errorp**: A boolean.
- **allow-remote**: A boolean.
- **max-number**: An integer in the inclusive range [1,254] or nil.
- **wait-reason**: A string or nil.
- **timeout**: A real number or nil.
- **wait-function**: A function of no arguments, or nil.
- **direction**: One of :io, :input, :output.
- **inherit-access-p**: A boolean.
- **access**: A list, keyword, integer or string.

**Values**
- **stream**: A stream or nil.
- **connectedp**: A boolean.
- **condition**: An error condition or nil.

**Description**
The function open-named-pipe-stream creates a stream that communicates via a named pipe, and then tries to establish a connection. For a connection to be established, another process (which can be a Lisp process in the same image, or another process altogether) must connect to it. In LispWorks this is done by connect-to-named-pipe, other software...
does by the underlying Windows function

ConnectNamedPipe.

open-named-pipe-stream returns three values. stream is a stream on success, and nil if there is an error and errorp is nil. If open-named-pipe-stream returns a stream and connectedp is non-nil, the stream is connected and is ready for input/output operations like a normal stream. condition is an error condition if an error occurred and errorp is nil, otherwise it is nil.

When open-named-pipe-stream returns a stream and connectedp is nil (which can happen when timeout is non-nil or wait-function returns t), the stream is valid but is not ready for I/O and gives an error for any reading or writing calls. In this case the function wait-for-connection must be used to establish the connection, and once it returns non-nil the stream is ready for input/output operations.

Note that that if you stop using a stream before it is connected, it still must be closed (by cl:close) to get rid of the named pipe.

The creation has two steps:

1. Creating the named pipe. This is controlled by name, max-number, direction, access, inherit-access-p, allow-remote and errorp.

2. Waiting for connection. This is controlled by timeout, wait-reason, and wait-function.

name is the pipe name. It can contain any character except #\ (according to the MSDN). open-named-pipe-stream prepends to it the pipe prefix \\.\pipe\. It needs to be highly unique, because on the same machine it is visible to all processes.

direction specifies the direction of I/O with the conventional meaning (as in Common Lisp file streams). The default value of direction is :io. All simultaneous opened pipes with the same name on the same machine must be opened with the
same value of direction. If different direction values are used, it causes `open-named-pipe-stream` to give an error.

`max-number` specifies the maximum number of simultaneously existing pipes with the same name on the local machine. By default it is unlimited. All simultaneous pipes with the same name on the same machine must have the same `max-number`, though currently this is not enforced.

`access` specifies access permission, which controls who can connect to the pipe. If it is `nil`, the permissions of the current thread are inherited and used (`inherit-access-p` is ignored in this case), and if `access` is `:world` the pipe is created with no restrictions. Otherwise `access` has to be a keyword, a list, an integer or a string, and it is passed to `security-description-string-for-open-named-pipe`. See the entry for `security-description-string-for-open-named-pipe` for details. The result of `security-description-string-for-open-named-pipe`, potentially with the inherited access appended, is passed to the Windows function `ConvertStringSecurityDescriptorToSecurityDescriptor` to generate the descriptor that is used when creating the pipe.

`inherit-access-p` controls whether the permissions of the current thread are inherited when `access` is not `nil` or `:world` or a string. When it is not `nil`, the permissions of the current thread are appended to what is specified by `access` (which means that the specification in `access` takes precedence). The default value of `inherit-access-p` is `t`.

`allow-remote` controls whether the pipe can be connected from another machine. The default value of `allow-remote` is `nil`.

`errorp` controls what happens when there is a failure because of one of these possibilities:
1. `security-description-string-for-open-named-pipe` returns `nil` because `access` contains unknown entities (for example a user name that does not exist on the local machine).

2. Converting the string that `security-description-string-for-open-named-pipe` returned to a security descriptor failed. That can happen if `access` is a string in bad format or a list containing strings in bad format.

3. `open-named-pipe-stream` failed for some other reason, for example it reached the limit on the number of the pipes with this name, or tried to open it with a different `direction` from the previous call.

When there has been a failure, if `errorp` is non-nil an error is signaled, and if `errorp` is `nil` then `open-named-pipe-stream` returns `stream nil` and `connectedp nil` with the error condition returned as the third value `condition`. The default value of `errorp` is `t`.

Once the pipe has been successfully created, `open-named-pipe-stream` uses `wait-for-connection` to establish the connection, passing `timeout`, `wait-reason` and `wait-function`, and returns the stream as first value, the result of `wait-for-connection` as the second value, and `nil` as the third value. See the description of `wait-for-connection` for details.

To connect to the other side of the pipe from Lisp, use `connect-to-named-pipe`. The Microsoft Windows function is `ConnectNamedPipe`. See also

- `wait-for-connection`
- `security-description-string-for-open-named-pipe`
- `named-pipe-stream-name`
- `connect-to-named-pipe`
- `impersonating-named-pipe-client`
This chapter applies only to LispWorks for Windows

**record-message-in-windows-event-log**

*Function*

**Summary**
Records a message in the Windows event log.

**Package**
`win32`

**Signature**
```
call-wind-record-message-in-windows-event-log type message &key
   source-name unc-server-name handle category event-id user-sid data
```

**Arguments**
- `type` A keyword.
- `message` A string or list of strings.
- `source-name` A string.
- `unc-server-name` `nil` or a string.
- `handle` `nil` or an open event log handle.
- `category` An integer.
- `event-id` An integer.
- `user-sid` `nil` or a foreign pointer to a SID object.
- `data` `nil` or a string.

**Description**
The function `record-message-in-windows-event-log` records a message `message` in the Windows event log.

- `type` must be one of the keywords `:success`, `:error`, `:warning`, `:information`, `:audit-success` or `:audit-failure`, corresponding to the types of Window event log entry.

- `message` is used as the string (or list of strings) recorded with the event.

If `handle` is `nil`, `source-name` is used as the name of the event source for recording events. If `source-name` is `nil` then the name of the Lisp executable is used.

If `handle` is `nil` and `unc-server-name` is non-nil, then it specifies the UNC name of a server which records the events.
If `handle` is non-nil, then it must be an open event log handle, such as created by `with-windows-event-log-event-source`. If `handle` is `nil`, then `source-name` is used to open an event log handle for the duration of the call to `record-message-in-windows-event-log`.

`category` and `event-id` are recorded in the event log. They are only useful if you create and register an event source provider DLL in Windows (see MSDN documentation for "Reporting Events").

If `user-sid` is non-nil, then it is used to record the user that logged the event.

If `data` is non-nil, then it is recorded as extra data associated with the event.

See also `with-windows-event-log-event-source`

---

**security-description-string-for-open-named-pipe**

*Function*

**Summary**

Interprets an access specification and generates a Security Descriptor String.

**Package**

`win32`

**Signature**

`security-description-string-for-open-named-pipe access-spec => result, fail-type, fail-item`

**Arguments**

- `access-spec` A keyword, an integer, a string or a list.

**Values**

- `result` A string or `nil`.
- `fail-type` Undefined, or a string.
- `fail-item` Undefined, or a keyword, an integer, a string or a list.
Description

The function `security-description-string-for-open-named-pipe` interprets `access-spec` and generates from it a Security Descriptor String as defined by Windows. See the MSDN for documentation of "Security Descriptor String Format".

`security-description-string-for-open-named-pipe` has quite limited capabilities, and the string that it generates contains only the DACL part of the descriptor.

If `access-spec` is a keyword, then its symbol name specifies a SID (Security Identifier). This SID gets read/write permission. The SID can be either standard representation (which looks like "S-1-...") or one of the aliases. The aliases are documented in the MSDN in the page titled "SID strings" (search for `SDDL_ANONYMOUS`). In general they have two letters, for example `AU` means authenticated users. The common standard strings are documented in the MSDN page titled "Well-known SIDs" (search for `SECURITY_WORLD_RID`). For example, `S-1-15-11` means authentication users. Any standard strings is acceptable, not only the documented ones, provided that it specifies a valid SID. For example, you can find the SID of a user by `user-name-to-sid-string`, intern it in the keyword package and use this (but it is better to pass a list `(:user)` as described below).

If `access-spec` is an integer, it must be one of the integers in the `WELL_KNOWN_SID_TYPE` Enumeration as documented in the MSDN. For example, 17 means authenticated users. The corresponding SID gets read/write permission.

If `access-spec` is a string, it is returned as-is. In this case it is the responsibility of the programmer to ensure that the string is valid. Note that if this string is used in `open-named-pipe-stream`, `open-named-pipe-stream` does not inherit the access even if `inherit-access-p` is non-nil.

If `access-spec` is a list, then each element in the list must be one of:
A string  The string must be a correct ACE (Access Control Entry) string, as described in the MSDN (search for "ACE strings"). The string must contain the opening and closing brackets, and can contain more than one ACE. 

`security-description-string-for-open-named-pipe` does not check the syntax in the string, and if the ACE is wrong the result string would be invalid.

A keyword  This is interpreted as when `access-spec` is a keyword, and the corresponding SID gets read/write permission.

An integer  This is interpreted as when `access-spec` is an integer, and the corresponding SID gets read/write permission.

A list of the form `(keyword SID-spec &rest permissions)`

The first element `keyword` specifies how to interpret the `SID-spec`. The possible keywords are: `:user`, when `SID-spec` must be a string and should name a user on the local machine, and `:sid`, when `SID-spec` must be a keyword, an integer or a string specifying the SID. Integers and keywords are interpreted as above, and strings are interpreted in the same way as keywords. If `permissions` are not supplied, they default to `(:read :write)`. When they are supplied, they are keywords specifying permission. Currently supported keywords are (i) one of `:read` or `:disallow-read` (ii) one of `:write` or `:disallow-write`, specifying the obvious meaning. It is an error if a keyword is repeated or if an incompatible pair is passed.

`security-description-string-for-open-named-pipe` returns 3 values. When successful, result is the string and the
other return values are undefined. When it fails, which can be because it is given an unrecognized SID specifier, result is \texttt{nil}, \texttt{fail-type} is a short string giving the type of the item that fails, and \texttt{fail-item} is the item in the list that fails when \texttt{access-spec} is a list.

Notes

1. When the argument is syntactically incorrect, \texttt{security-description-string-for-open-named-pipe} signals an error. It fails and returns \texttt{nil} only when a SID specifier of an acceptable type does not specify a SID.

2. Except when given a string which is returned as-is, \texttt{security-description-string-for-open-named-pipe} works by generating an ACE (Access Control Entry) string for each SID giving it the read and write permission, except in the case when either \texttt{:disallow-write} or \texttt{:disallow-read} is used, when it generates an ACE string denying permission. All the ACEs are then concatenated and \texttt{"D:"} is prepended, thus generating a Security Descriptor String containing only the DACL.

3. Experimentally, you can connect to a named pipe only if you have both read and write permission, even when opening it for only read or only write. Thus when this function is used from \texttt{open-named-pipe-stream}, the keywords \texttt{:disallow-read} etc are not very useful. They are useful only when you want to deny access for a specific SID, by using \texttt{:disallow-read} and \texttt{:disallow-write}.

4. The order of the items in the list is significant: earlier items override later items.

5. Disallowing permission, for example by using \texttt{:disallow-read}, is not the same as not allowing it, because in the latter case a later ACE can give the SID the permission. Disallowing prevents later ACEs from giving permission.
6. Using a string or ACE strings in the list allows the user to generate a more elaborate string than security-description-string-for-open-named-pipe knows how to generate. In this case the returned string may be invalid. When this happens from open-named-pipe-stream, open-named-pipe-stream will get a failure and signal or return an error according to errorp.

Examples
Any of these gives permissions to all authenticated users:

```
:AU
17
':(:AU)
'(17)
'((:SID :AU))
'(:(SID "AU")
'((:SID 17))
```

Also, all of the above with AU replaced by S-1-15-11 will give permission to all authenticated users.

This gives permissions to all authorized users except the user "exclude":

```
'(((use "exclude" :DISALLOW-READ :DISALLOW-WRITE)
:AU)
```

See also
canonicalize-sid-string
named-pipe-stream-name
open-named-pipe-stream
sid-string-to-user-name
user-name-to-sid-string

**set-application-themed**

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
</tr>
<tr>
<td><strong>Package</strong></td>
</tr>
<tr>
<td><strong>Signature</strong></td>
</tr>
</tbody>
</table>
This chapter applies only to LispWorks for Windows

<table>
<thead>
<tr>
<th>Arguments</th>
<th>on/off</th>
<th>A generalized boolean.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The function \texttt{set-application-themed} controls whether a LispWorks application should be themed. On supported versions of Microsoft Windows, LispWorks is &quot;themed&quot;, that is it uses the current theme of the desktop. You can switch this off by calling \texttt{(win32:set-application-themed nil)}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>On systems older than Windows XP (no longer supported), or when the application does not have Common Controls 6, this call has no effect. \texttt{set-application-themed} affects only windows that are created after it was called. Normally, it should be called before any window is created, so all LispWorks windows will appear with the same theme. However, \texttt{set-application-themed} can be called multiple times in the same run.</td>
<td></td>
</tr>
</tbody>
</table>

\textbf{short-namestring} \hfill \textit{Function}

| Summary | Returns the short form of a namestring. |
| Package | \texttt{win32} |
| Signature | \texttt{short-namestring pathname => result} |
| Arguments | \texttt{pathname} \quad A pathname designator. |
| Values | \texttt{result} \quad A string or \texttt{nil}. |
| Description | The function \texttt{short-namestring} first obtains the full namestring as if by \texttt{cl:namestring}, and then converts this namestring to the short form (in the Microsoft Windows meaning of "Short" paths). |
If the translation succeeds then result is a string in the short form.
The translation may fail, in which case nil is returned.

See also long-namestring

**sid-string-to-user-name**

*Function*

Summary Takes a standard SID (Security Identifier) string and locates the user.

Package win32

Signature sid-string-to-user-name sid-string => result

Arguments sid-string A string.

Values result A string or nil.

Description The function sid-string-to-user-name takes a standard SID (Security Identifier) string and tries to locate the user. It returns nil if sid-string is not the SID of a user.

See also wait-for-connection

security-description-string-for-open-named-pipe

**str**  
**lpcstr**  
**lpstr**

*FLI type descriptors*

Summary Types converting to ANSI strings.

Package win32
This chapter applies only to LispWorks for Windows

Signature

\begin{verbatim}
str &key length
lpstr &key max-length
lpctstr &key max-length
\end{verbatim}

Description

\begin{verbatim}
str is an ANSI string.
lpctstr is a reference-pass pointer to an ANSI string.
lpstr is a reference (in/out) pointer to an ANSI string.
These types are ANSI only. Use these if you do not need the power of Unicode. Take care to interface to ANSI functions named like FooBarA, with the A suffix.
\end{verbatim}

See also

\begin{verbatim}
tstr
lpctstr
lptstr
\end{verbatim}

**tstr**

**lpctstr**

**lptstr**

\textit{FLI type descriptors}

Summary

Types which automatically switch between ANSI and Unicode strings.

Package

\begin{verbatim}
win32
\end{verbatim}

Signature

\begin{verbatim}
tstr &key length
lpctstr &key max-length
lptstr &key max-length
\end{verbatim}

Description

\begin{verbatim}
tstr is an ANSI/Unicode string.
lpctstr is a reference-pass pointer to ANSI/Unicode string.
lptstr is a reference (in/out) pointer to an ANSI/Unicode string.
\end{verbatim}
Each of these three types automatically switch between ANSI and Unicode, which makes them ideal for use with the :dbcs encoding option in fli:define-foreign-function.

Example

This calls GetDriveTypeA on Windows ME, and GetDriveTypeW on supported versions of Windows.

The argument is passed as ANSI or Unicode respectively:

```lisp
(fli:define-foreign-function (%get-drive-type
 "GetDriveType" :dbcs)
   ((lpRootPathName W:LPCTSTR))
 :result-type (:unsigned :int))

(defconstant +drive-types+

(defun get-drive-information (drive)
   (the drive-type (svref +drive-types+ (%get-drive-type drive))))
```

**wstr**

**lpcwstr**

**lpwstr**

*FLI type descriptors*

**Summary**

Types converting to Unicode strings.

**Package**

win32

**Signature**

```
wstr &key length
lpcwstr &key max-length
lpwstr &key max-length
```

**Description**

* wstr is a Unicode string.
* lpcwstr is a reference-pass pointer to a Unicode string.
* lpwstr is a reference (in/out) pointer to a Unicode string.
These three types are Unicode only.

See also  \texttt{tstr}

\textbf{user-name-to-sid-string} \hspace{1cm} \textit{Function}

\textbf{Summary} \hspace{1cm} Returns a standard SID (Security Identifier) associated with the user.

\textbf{Package} \hspace{1cm} \texttt{win32}

\textbf{Signature} \hspace{1cm} \texttt{user-name-to-sid-string user-name => sid-string}

\textbf{Arguments} \hspace{1cm} \texttt{user-name} \hspace{1cm} A string.

\textbf{Values} \hspace{1cm} \texttt{sid-string} \hspace{1cm} A string or \texttt{nil}.

\textbf{Description} \hspace{1cm} The function \texttt{user-name-to-sid-string} returns a standard SID (Security Identifier) associated with the user \texttt{user-name} on the current machine. It returns \texttt{nil} if it failed to find the user.

See also \hspace{1cm} \texttt{wait-for-connection}

\texttt{security-description-string-for-open-named-pipe}

\textbf{wait-for-connection} \hspace{1cm} \textit{Generic Function}

\textbf{Summary} \hspace{1cm} Waits to establish a connection for a stream.

\textbf{Package} \hspace{1cm} \texttt{win32}

\textbf{Signature} \hspace{1cm} \texttt{wait-for-connection stream &key timeout wait-reason wait-function => connectedp}

\textbf{Arguments} \hspace{1cm} \texttt{stream} \hspace{1cm} A named pipe stream.
timeout A non-negative real number, or nil.

wait-reason A string, or nil.

wait-function A function designator, or nil.

Values

connectedp A generalized boolean.

Description

The function wait-for-connection waits until it succeeds to establish a connection for the stream stream, or timeout seconds passed or wait-function returns non-nil, and returns a value indicating whether the connection is established successfully.

stream must be a stream of the right type. Currently the only supported stream is a named pipe stream (the result of open-named-pipe-stream).

timeout can be nil or a real number specifying the time in seconds before wait-for-connection returns without establishing a connection.

wait-reason, if non-nil, needs to be a string specifying the wait reason. It has the same semantics as the wait-reason argument of process-wait.

wait-function, if non-nil, must be a function of no arguments. If it returns non-nil, wait-for-connection returns nil.

wait-for-connection can be repeatedly called on the same stream. If the stream has already established a connection, it returns true immediately.

Notes

wait-function has the same limitations as the wait-function of process-wait.

See also open-named-pipe-stream
This chapter applies only to LispWorks for Windows

**with-windows-event-log-event-source**  
*Macro*

**Summary**  
Provides an open event log handle for a body of code.

**Package**  
`win32`

**Signature**  

```
with-windows-event-log-event-source (handle source-name
 &optional unc-server-name) &body body => values
```

**Arguments**

- `handle`  
  A symbol.

- `source-name`  
  `nil` or a string.

- `unc-server-name`  
  `nil` or a string.

**Values**

- `values`  
  The values returned by `body`

**Description**  
The macro `with-windows-event-log-event-source` provides an open event log handle for a body of code.

The macro `with-windows-event-log-event-source` binds `handle` to an open event log handle, evaluates the forms of `body` and closes `handle`. The values of the last form in `body` are returned.

`source-name` is used as the name of the event source for recording events. If `source-name` is `nil` then the name of the Lisp executable is used.

If `unc-server-name` is non-nil, then it specifies the UNC name of a server which records the events.

**See also**  
`record-message-in-windows-event-log`
This chapter describes the Microsoft Windows registry API, which is available in the WIN32 package.

The WIN32 package also includes “Miscellaneous WIN32 symbols”, “The DDE client interface” and “The DDE server interface”. These are documented in separate chapters in this manual.

**Note:** the WIN32 package is not a supported implementation of the Win32 API. You should not use symbols in the WIN32 package unless they are documented in this manual. Instead, define your own interfaces to Windows functions as you need - see the LispWorks Foreign Language Interface User Guide and Reference Manual for details.

**Note:** this chapter applies only to LispWorks for Windows, and not the UNIX, Linux, x86/x64 Solaris, FreeBSD, AIX or Mac OS X platforms.

---

**close-registry-key**

**Summary** Closes a handle to an open registry key.

**Package** win32
Signature: `close-registry-key handle &key errorp => successp, error-code`

Arguments:
- `handle` A handle to an open registry key.

Values:
- `successp` A boolean.
- `error-code` An integer error code or `nil`.

Description:
The function `close-registry-key` closes `handle`, which should be an open registry key handle. The return value on success is `t`. If an error occurs and `errorp` is true then an error is signaled. Otherwise, the return values are `nil` and the Windows `error-code`. The default value of `errorp` is `t`.

See also:
- `create-registry-key`
- `open-registry-key`

---

**collect-registry-subkeys**

*Function*

Summary:
Returns names of the subkeys of a registry key.

Package: `win32`

Signature: `collect-registry-subkeys subkey &key root max-name-size max-names errorp value-function => subsubkeys`

Arguments:
- `subkey` A string specifying the name of the key.
- `root` A keyword or handle.
- `max-name-size` An integer.
- `max-names` An integer.
- `errorp` A boolean.
- `value-function` A function designator or `nil`. 
Values

| subkeys | A list. |

Description

The function `collect-registry-subkeys` returns a list of names which are subsubkeys of `subkey` under the key `root`.

`subkey` and `root` are interpreted as described for `create-registry-key`. The default value of `root` is `:user`.

`max-name-size` specifies the maximum length of the returned name. If the name is longer than this, an error is signaled. The default value of `max-name-size` is 256.

`max-names` specifies the maximum number of names returned. Names after this number are ignored. The default value of `max-names` is `most-positive-fixnum`.

If `value-function` is non-nil, it should be a function with signature

```lisp
(value-function handle subkey-name => name, collectp)
```

`value-function` is funcalled for each subsubkey with the handle of `subkey` and the name of the subsubkey. If `collectp` is non-nil then `name` is collected into the list `subsubkeys` to return from `collect-registry-subkeys`. Otherwise it is ignored.

If `value-function` is `nil`, then the returned `subsubkeys` is a list of strings naming all (subject to `max-names`) of the subsubkeys. The default value of `value-function` is `nil`.

If an error occurs opening `subkey` and `errorp` is `true` then an error is signaled. Otherwise, `subsubkeys` is returned as `nil` if `subkey` could not be opened. The default value of `errorp` is `t`.

See also

`collect-registry-values`

`create-registry-key`

---

**collect-registry-values**

**Function**

**Summary**

Returns the values of a registry key.
Package  win32

Signature collect-registry-values subkey &key root max-name-size max-buffer-size expected-type errorp value-function => values-alist

Arguments subkey A string specifying the name of the key.
root A keyword or handle.
max-name-size An integer.
max-buffer-size An integer.
expected-type A keyword or t.
errorp A boolean.
value-function A function or symbol.

Values values-alist An alist.

Description The function collect-registry-values returns an alist of all of the values of subkey under the key root.

subkey and root are interpreted as described for create-registry-key. The default value of root is :user.

max-name-size specifies the maximum length of the returned name. If the name is longer than this, an error is signaled. The default value of max-name-size is 256.

max-buffer-size specifies the maximum length in bytes of the data. If the data is longer than this, an error is signaled. The default value of max-buffer-size is 1024.

If value-function is nil, the returned values-alist is an association list containing pairs (name . data) consisting of the names and data of the values of subkey. expected-type controls how certain types are converted to Lisp objects as described for enum-registry-value. The default value of expected-type is t.

If value-function is non-nil, it should be a function with signature
value-function handle subskey-name-and-value => name-and-value, collectp

value-function is funcalled for each subskey with the handle of subskey and a cons of the name and value of the subskey. If collectp is non-nil then name-and-value is collected into the alist values-alist to return from collect-registry-values. Otherwise name-and-value is ignored.

If an error occurs and errorp is true, then an error is signaled. Otherwise, values-alist is returned as nil if subskey could not be opened at all or contains nil for the data of any particular pair that cannot be read. The default value of errorp is t.

See also

collect-registry-subkeys
create-registry-key
enum-registry-value

create-registry-key

Function

Summary Creates a new registry key.

Package win32

Signature create-registry-key subskey &key class root access volatile errorp => handle, disposition, error-code

Arguments

subkey A string specifying the name of the key.
class A string.
root A keyword or handle.
access A keyword or an integer.
volatile A generalized boolean.
errorp A generalized boolean.

Values

handle The handle of the new key.
The function `create-registry-key` creates a new registry key named `subkey` under the parent key `root`. If the key already exists, it is opened and returned.

`subkey` is a string specifying a path from a root. Each component of the path is separated by a backslash. Use `""` to denote the null path (that is, the root).

`class` can be used to specify the class of the key if it is created.

`root` should be a handle to an open registry key (for example a key returned by `create-registry-key` or `open-registry-key` or one of the keywords `:classes`, `:user`, `:local-machine` and `:users` which represent the standard top level roots in the registry. The default value of `root` is `:user`.

If `access` is `:read`, then the key is created with `KEY_READ` permissions. If `access` is `:write`, then the key is created with `KEY_WRITE` permissions. If `access` is an integer, then the value `access` specifies the desired Win32 access rights. The default value of `access` is `:read`.

The return values on success are the handle of the new key and a keyword `:created-new-key` or `:open-existing-key` indicating whether a new key was created or opened.

When `volatile` is true, the key is created with `REG_OPTION_VOLATILE`. `volatile` defaurs to false.

If an error occurs and `errorp` is true then an error is signaled. Otherwise, the return values are `nil`, `nil` and the Windows `error-code`. The default value of `errorp` is `t`.

See also
- `delete-registry-key`
- `open-registry-key`
This chapter applies only to LispWorks for Windows

**delete-registry-key**

**Function**

Summary: Deletes a registry key.

Package: `win32`

Signature: `delete-registry-key subkey &key root errorp => successp, error-code`

Arguments:
- `subkey`: A string specifying the name of the key.
- `root`: A keyword or handle.
- `errorp`: A generalized boolean.

Values:
- `successp`: A boolean.
- `error-code`: An integer error code or `nil`.

Description: The function `delete-registry-key` deletes the registry key named `subkey` under the parent key `root`.

`subkey` and `root` are interpreted as described for `create-registry-key`. The default value of `root` is `:user`.

The value `t` is returned if the key is deleted successfully.

If an error occurs and `errorp` is true then an error is signaled. Otherwise, the return values are `nil` and the Windows `error-code`. The default value of `errorp` is `t`.

See also: `create-registry-key`

**enum-registry-value**

**Function**

Summary: Enumerates the values of a registry key.

Package: `win32`
The function `enum-registry-value` allows the values of subkey under the key `root` to be enumerated.

`subkey` and `root` are interpreted as described for `create-registry-key`. The default value of `root` is `:user`.

`index` specifies which value to return, with 0 being the first item.

`max-name-size` specifies the maximum length of the returned name. If the name is longer than this, an error is signaled. The default value of `max-name-size` is 256.

`max-buffer-size` specifies the maximum length in bytes of the value. The value is longer than this, an error is signaled. The default value of `max-buffer-size` is 1024.

If the value exists (that is, `index` is not too large), then the return values are the name, data type and data associated with the value in the registry. The argument `expected-type` con-
This chapter applies only to LispWorks for Windows

trols how certain data types are converted to Lisp objects as follows:

Table 51.1 Conversion of registry values to Lisp objects

<table>
<thead>
<tr>
<th>data-type</th>
<th>expected-type</th>
<th>Description of converted data</th>
</tr>
</thead>
<tbody>
<tr>
<td>:string</td>
<td>:lisp-object</td>
<td>String made with <code>read-from-string</code></td>
</tr>
<tr>
<td>:string</td>
<td>Not supplied</td>
<td>String, exactly as in the registry</td>
</tr>
<tr>
<td>:environment-string</td>
<td>:string</td>
<td>String, exactly as in the registry</td>
</tr>
<tr>
<td>:environment-string</td>
<td>Not supplied</td>
<td>String, environment variables expanded</td>
</tr>
<tr>
<td>:integer</td>
<td>Not supplied</td>
<td>Integer</td>
</tr>
<tr>
<td>:little-endian-integer</td>
<td>Not supplied</td>
<td>Integer</td>
</tr>
<tr>
<td>:binary</td>
<td>Not supplied</td>
<td>A newly allocated foreign object</td>
</tr>
<tr>
<td>:binary</td>
<td>:lisp-object</td>
<td>Vector, element type (unsigned-byte 8)</td>
</tr>
</tbody>
</table>

The default value of `expected-type` is t.

If an error occurs and `errorp` is true, then an error is signaled. Otherwise, the return values are nil, nil, nil and the Windows `error-code`. The default value of `errorp` is t.

See also `create-registry-key`
open-registry-key

Function

Summary
Opens a registry key.

Package
win32

Signature
open-registry-key subkey &key root access errorp => handle, error-code

Arguments
subkey A string specifying the name of the key.
root A keyword or handle.
access An integer or keyword.
errorp A generalized boolean.

Values
handle The handle of the key.
error-code An integer error code or nil.

Description
The function open-registry-key opens a registry key named subkey under the parent key root.

subkey and root are interpreted as described for create-registry-key. If subkey is an empty string, then the root key is returned. The default value of root is :user.

If access is :read, then it opens the key with KEY_READ permissions. If access is :write, then it opens the key with KEY_WRITE permissions. If access is an integer, then the value access specifies the desired Win32 access rights. If access is omitted and root is :user, then open-registry-key uses KEY_ALL_ACCESS. Otherwise it uses KEY_READ.

The return value on success is the handle of the opened key.

If an error occurs and errorp is true, then an error is signaled. Otherwise, the return values are nil and the Windows error-code. The default value of errorp is t.

See also create-registry-key
query-registry-key-info  

**Summary**  
Returns information about an open registry key handle.

**Package**  
win32

**Signature**  
query-registry-key-info key => info, error-code

**Arguments**  
key  
A handle.

**Values**  
info  
A property list.

error-code  
An integer error code or nil.

**Description**  
The function query-registry-key-info returns a plist of information about the open registry key handle key. The elements of the plist info are:

: class  
A string naming the class of the key, if any.

: subkeys-count  
An integer giving the number of subkeys.

: subkey-max-len  
An integer giving the length of the longest subkey name.

: class-name-max-len  
An integer giving the length of the longest class name.

: values-count  
An integer giving the number of values.

: value-max-len  
An integer giving the length of the longest value name.

: max-data-len  
An integer giving the length of the longest value data.

: security-len  
An integer giving the length of the security descriptor.
query-registry-value

Function

Returns a value stored in the registry.

Package

win32

Signature

query-registry-value subkey name &key root expected-type
errorp => data, successp, error-code

Arguments

subkey A string specifying the name of the key.
name A string specifying the name of the value.
root A keyword or handle.
expected-type A keyword or t.
errorp A boolean.

Values

data A Lisp object.
successp A boolean.
error-code An integer error code or nil.

Description

The function query-registry-value returns the value associated with name in subkey under the key root.

subkey and root are interpreted as described for create-registry-key. If subkey is an empty string, then the root key is returned. The default value of root is :user.

If the value exists, then the return values are the data and true. expected-type controls how certain types are converted to the Lisp object data as described for enum-registry-value. The default value of expected-type is t.

If an error occurs and errorp is true then an error is signaled. Otherwise, the return values are nil, nil and the Windows error-code. The default value of errorp is t.

See also

create-registry-key
enum-registry-value
This chapter applies only to LispWorks for Windows

**registry-key-exists-p**

**Function**

**Summary**
The predicate for whether a registry key can be opened.

**Package**
win32

**Signature**
registry-key-exists-p subkey &key root access => existsp

**Arguments**
- subkey: A string specifying the name of the key.
- root: A keyword or handle.
- access: An integer or keyword.

**Values**
- existsp: A boolean.

**Description**
The function `registry-key-exists-p` checks whether the registry key named `subkey` can be opened under the parent key `root` with the supplied `access` permissions.

`subkey` and `root` are interpreted as described for `create-registry-key`. The default value of `root` is `:user`.

If `access` is `:read`, then it opens the key with `KEY_READ` permissions. If `access` is `:write`, then it opens the key with `KEY_WRITE` permissions. If `access` is an integer, then the value `access` specifies the desired Win32 access rights. If `access` is omitted and `root` is `:user`, then `registry-key-exists-p` uses `KEY_ALL_ACCESS`. Otherwise it uses `KEY_READ`.

`registry-key-exists-p` closes the key before returning, but the return value is `t` if the key could actually be opened and `nil` otherwise.

**See also**
create-registry-key

**registry-value**

**Accessor**

**Summary**
Gets or sets a value in the registry.
Package win32

Signature

registry-value subkey name &key root expected-type errorp =>
data, successp, error-code

(setf registry-value) value subkey name &key root expected-
type errorp => value

Arguments

subkey A string specifying the name of the key.
name A string specifying the name of the value.
root A keyword or handle.
expected-type A keyword or t.
errorp A boolean.

Values

data A Lisp object.
successp A boolean.
error-code An integer error code or nil.

Description

The function registry-value returns the value associated with name in subkey under the key root.

subkey and root are interpreted as described for create-registry-key. The default value of root is :user.

If the value exists, then the return values are the data and true. expected-type controls how certain types are converted to Lisp objects as described for enum-registry-value. The default value of expected-type is t.

If an error occurs and errorp is true then an error is signaled. Otherwise, the return values are nil, nil and the Windows error-code. The default value of errorp is t.

The function (setf registry-value) sets the value associated with name in subkey under the key root, creating the subkey if necessary. The default value of root is :user.

See also set-registry-value
**set-registry-value**  
*Function*

Summary  
Stores a value in the registry.

Package  
win32

Signature  
```
set-registry-value data subkey name &key root expected-type errorp => error-code
```

Arguments  
- **data**  
  A Lisp object.
- **subkey**  
  A string specifying the name of the key.
- **name**  
  A string specifying the name of the value.
- **root**  
  A keyword or handle.
- **expected-type**  
  A keyword or t.
- **errorp**  
  A boolean.

Values  
- **error-code**  
  An integer error code or `nil`.

Description  
The function `set-registry-value` sets the value associated with `name` in `subkey` under the key `root`.

`subkey` and `root` are interpreted as described for `create-registry-key`. The default value of `root` is `:user`.

The stored value is derived from `data`, converted according to `expected-type` as follows:

<table>
<thead>
<tr>
<th>Lisp data</th>
<th>expected-type</th>
<th>Registry type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A string</td>
<td>:string</td>
<td>REG_SZ exactly as in <code>data</code></td>
</tr>
<tr>
<td>Lisp value</td>
<td>:lisp-object</td>
<td>REG_SZ made with <code>prin1-to-string</code> of <code>data</code></td>
</tr>
</tbody>
</table>

1937
The default value of `expected-type` is `t`.

If an error occurs and `errorp` is true then an error is signaled. The default value of `errorp` is `t`.

See also `create-registry-key`
`registry-value`

### Table 51.2 Conversion of Lisp objects to registry values

<table>
<thead>
<tr>
<th>Lisp data</th>
<th><code>expected-type</code></th>
<th>Registry type</th>
</tr>
</thead>
<tbody>
<tr>
<td>An integer</td>
<td>:integer</td>
<td>REG_DWORD containing data</td>
</tr>
<tr>
<td>A foreign pointer</td>
<td>:binary</td>
<td>REG_BINARY containing bytes of one element at the pointer</td>
</tr>
<tr>
<td>An array</td>
<td>:binary</td>
<td>REG_BINARY containing bytes from the array</td>
</tr>
</tbody>
</table>

The default value of `expected-type` is `t`.

with-registry-key

**Macro**

**Summary**
Runs code with an open registry key handle.

**Package**
`win32`

**Signature**
```
(with-registry-key (handle subkey &key root access errorp)
 &body body => values)
```

**Arguments**
- `handle`  A variable name.
- `subkey`  A string specifying the name of the key.
- `root`  A keyword or handle.
- `access`  An integer or keyword.
errorp  A boolean.

Values  values  The values returned by body.

Description  The macro with-registry-key evaluates body with the variable handle bound to the registry key handle opened as if by calling

(open-registry-key subkey :root root
 :access access
 :errorp errorp)

subkey and root are interpreted as described for create-registry-key.

If errorp is nil and subkey cannot be opened then body is not evaluated.

See also  create-registry-key
This chapter applies only to LispWorks for Windows
The DDE client interface

This chapter describes the Dynamic Data Exchange (DDE) client interface which is available in the WIN32 package. You should use this chapter in conjunction with Chapter 22, “Dynamic Data Exchange”.

The WIN32 package also includes “Miscellaneous WIN32 symbols”, “The Windows registry API” and “The DDE server interface”. These are documented in separate chapters in this manual.

Note: the WIN32 package is not a supported implementation of the Win32 API. You should not use symbols in the WIN32 package unless they are documented in this manual. Instead, define your own interfaces to Windows functions as you need - see the LispWorks Foreign Language Interface User Guide and Reference Manual for details.

Note: this chapter applies only to LispWorks for Windows, and not the UNIX, Linux, x86/x64 Solaris, FreeBSD, AIX or Mac OS X platforms.

**dde-advise-start**

*Function*

**Summary** Sets up an advise loop on a specified data item for a conversation.
The DDE client interface

This chapter applies only to LispWorks for Windows

<table>
<thead>
<tr>
<th>Package</th>
<th>win32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>\texttt{dde-advice-start conversation item &amp;key key function format datap type errorp =&gt; result}</td>
</tr>
<tr>
<td>Arguments</td>
<td>conversation: A conversation object. \item item: A string or symbol. \item key: An object. \item function: A function name. \item format: A clipboard format specifier. \item datap: A boolean. \item type: A keyword. \item errorp: A boolean.</td>
</tr>
<tr>
<td>Values</td>
<td>result: A boolean.</td>
</tr>
<tr>
<td>Description</td>
<td>The function \texttt{dde-advice-start} sets up an advise loop for the data item specified by \texttt{item} on the specified \texttt{conversation}. See “Advise loops” on page 324 for information about DDE advise loops. The argument \texttt{format} should be one of the following: \begin{itemize} \item A DDE format specifier, consisting of either a standard clipboard format or a registered clipboard format. \item A string containing either the name of a standard clipboard format (without the \texttt{CF} prefix), or the name of a registered clipboard format. \item A symbol, in which case its print name is taken to specify the clipboard format. \item The keyword \texttt{:text} – the default value of \texttt{format}. The keyword \texttt{:text} is treated specially. If supported by the server it uses the \texttt{CF_UNICODETEXT} clipboard format, otherwise it used the \texttt{CF_TEXT} format. \end{itemize}</td>
</tr>
</tbody>
</table>
The argument type specifies how the response data should be converted to a Lisp object. For text formats, the default value indicates that a Lisp string should be created. The value :string-list may be specified to indicate that the return value should be taken as a tab-separated list of strings; in this case the Lisp return value is a list of strings. The default conversation class only supports text formats, unless type is specified as :foreign, which can be used with any clipboard format. It returns a clipboard-item structure, containing a foreign pointer to the data, the data length, and the format identifier.

If datap is t (the default value), a hot link is established, where the new data is supplied whenever it changes. If datap is nil, a warm link is established, where the data is not passed, and must be explicitly requested using dde-request.

The argument key is used to identify this link. If specified as nil (the default value), it defaults to the conversation. Multiple links are permitted on a conversation with the same item and format values, as long as their key values differ.

If the link is established, the return value result is t. If the link could not be established, the behavior depends on the value of errorp. If errorp is t (the default value), LispWorks signals an error. If it is nil, the function returns nil to indicate failure.

If the link is established, the function function is called whenever the data changes. If function is nil (the default value), then the generic function dde-client-advise-data will be called.

The function specified by function should have a lambda list similar to the following:

key item data &key conversation &allow-other-keys

The arguments key and item identify the link. The argument data contains the new data for hot links; for warm links it is nil.
See also  
dde-advise-start*  
dde-advise-stop  
dde-client-advise-data

**dde-advise-start**

*Function*

**Summary**
Sets up an advise loop for a specified data item for an automatically managed conversation.

**Package**
win32

**Signature**

dde-advise-start* service topic item &key key function format datap type errorp connect-error-p new-conversation-p => result

**Arguments**
- *service*  
  A string or symbol.
- *topic*  
  A string or symbol.
- *item*  
  A string or symbol.
- *key*  
  An object.
- *function*  
  A function name.
- *format*  
  A clipboard format specifier.
- *datap*  
  A boolean.
- *type*  
  A keyword.
- *errorp*  
  A boolean.
- *connect-error-p*  
  A boolean.
- *new-conversation-p*  
  A boolean.

**Values**
- *result*  
  A boolean.

**Description**
The function **dde-advise-start** is similar to the **dde-advise-start**, and sets up an advise loop for the data item.
specified by item on a conversation recognizing the service/topic pair.

See `dde-advertise-start` for information on DDE advise loops and the format, type, and datap arguments.

The argument key is used to identify this link. If specified as `nil` (the default value), it defaults to the conversation. Multiple links are permitted on a conversation with the same item and format values, as long as their key values differ.

If the link is established, the return value result is t. If the link could not be established, the behavior depends on the value of `errorp`. If `errorp` is t (the default value), LispWorks signals an error. If it is `nil`, the function returns `nil` to indicate failure.

If the link is established, the function function will be called whenever the data changes. If function is `nil` (the default value), the generic function `dde-client-advertise-data` will be called.

The function specified by function should have a lambda list similar to the following:

```
key item data &key conversation &allow-other-keys
```

The arguments key and item identify the link. The argument data contains the new data for hot links; for warm links it is `nil`.

See also

- `dde-advertise-start`
- `dde-advertise-stop`
- `dde-advertise-stop*`
- `dde-client-advertise-data`

### dde-advertise-stop

**Function**

**Summary**

Removes a link from a conversation specified by a given item and key.
<table>
<thead>
<tr>
<th>Package</th>
<th>win32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td><code>dde-advice-stop conversation item &amp;key key format errorp disconnectp no-advice-ok =&gt; result</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>conversation</code> A conversation object.</td>
</tr>
<tr>
<td></td>
<td><code>item</code> A string or symbol.</td>
</tr>
<tr>
<td></td>
<td><code>key</code> An object.</td>
</tr>
<tr>
<td></td>
<td><code>format</code> A clipboard format specifier.</td>
</tr>
<tr>
<td></td>
<td><code>errorp</code> A boolean.</td>
</tr>
<tr>
<td></td>
<td><code>disconnectp</code> A boolean.</td>
</tr>
<tr>
<td></td>
<td><code>no-advice-ok</code> A boolean.</td>
</tr>
<tr>
<td>Values</td>
<td><code>result</code> A boolean.</td>
</tr>
<tr>
<td>Description</td>
<td>The function <code>dde-advice-stop</code> removes a particular link from <code>conversation</code> specified by <code>item</code>, <code>format</code> and <code>key</code>. If <code>key</code> is the last key for the <code>item/format</code> pair, the advise loop for the pair is terminated.</td>
</tr>
<tr>
<td></td>
<td>See “Advise loops” on page 324 for information about DDE advise loops.</td>
</tr>
<tr>
<td></td>
<td>If <code>disconnectp</code> is <code>t</code>, and the last advise loop for the conversation is terminated, the conversation is disconnected.</td>
</tr>
<tr>
<td></td>
<td>Attempting to remove a link that does not exist raises an error, unless <code>no-advice-ok</code> is <code>t</code>.</td>
</tr>
<tr>
<td></td>
<td>If this function succeeds, it returns <code>t</code>. If it fails, the behavior depends on the value of <code>errorp</code>. If <code>errorp</code> is <code>t</code> (the default value), LispWorks signals an error. If <code>errorp</code> is <code>nil</code>, the function returns <code>nil</code> to indicate failure.</td>
</tr>
<tr>
<td>See also</td>
<td><code>dde-advice-start</code></td>
</tr>
<tr>
<td></td>
<td><code>dde-advice-start*</code></td>
</tr>
</tbody>
</table>
This chapter applies only to LispWorks for Windows

**dde-advise-stop***
**dde-client-advise-data**

**dde-advise-stop***  
*Function*

**Summary**  
Removes a link from an automatically managed conversation specified by a given item and key.

**Package**  
win32

**Signature**  
dde-advise-stop* service topic item &key key format errorp disconnectp => result

**Arguments**  
service  
A string or symbol.

topic  
A string or symbol.

item  
A string or symbol.

key  
An object.

format  
A clipboard format specifier.

errorp  
A boolean.

disconnectp  
A boolean.

**Values**  
result  
A boolean.

**Description**  
The function dde-advise-stop* is similar to the function dde-advise-stop, and removes a particular link from a conversation specified by the service/topic pair indicated by item, format and key. If key is the last key for the item/format pair, the advise loop for the pair is terminated.

See “Advise loops” on page 324 for information about DDE advise loops.

If disconnectp is t (the default value), and the last advise loop for the conversation is terminated, the conversation is disconnected.
If this function succeeds, it returns \texttt{t}. If it fails, the behavior depends on the value of \texttt{errorp}. If \texttt{errorp} is \texttt{t} (the default value), LispWorks signals an error. If \texttt{errorp} is \texttt{nil}, the function returns \texttt{nil} to indicate failure.

See also: \texttt{dde-advise-start} \hfill \texttt{dde-advise-stop}  
\texttt{dde-advise-start*}  
\texttt{dde-client-advise-data}

\textbf{dde-client-advise-data} \hspace{1cm} \textit{Generic Function}

\textbf{Summary} \hfill Called when data changes in an advise loop.

\textbf{Package} \hfill \texttt{win32}

\textbf{Signature} \hfill \texttt{dde-client-advise-data key item data &key &allow-other-keys}

\textbf{Arguments} \hfill
\begin{itemize}
  \item \texttt{key} \hspace{1cm} An object.
  \item \texttt{item} \hspace{1cm} A string or symbol.
  \item \texttt{data} \hspace{1cm} A string.
\end{itemize}

\textbf{Values} \hfill None.

\textbf{Description} \hfill The generic function \texttt{dde-client-advise-data} is the default function called when an advise loop informs a client that the data monitored by the loop has changed. By default it does nothing, but it may be specialized on the object used as the key in \texttt{dde-advise-start} or \texttt{dde-advise-start*}, or on a client conversation class if the default \texttt{key} is used.

See “Advise loops” on page 324 for information about DDE advise loops.

See also: \texttt{dde-advise-start} \hfill \texttt{dde-advise-stop}
This chapter applies only to LispWorks for Windows

**dde-connect**

*Function*

**Summary**
Attempts to create a conversation with a specified DDE server.

**Package**
win32

**Signature**
dde-connect service topic &key class errorp => object

**Arguments**
- `service` A symbol or string.
- `topic` A symbol or string.
- `class` The class of the conversation object to create.
- `errorp` A boolean.

**Values**
- `object` A conversation object.

**Description**
The function `dde-connect` attempts to create a conversation with a DDE server. If `server` names a client service registered with `define-dde-client`, the registered service name is used as the DDE service name. If `server` is any other symbol, the print name of the symbol is used as the DDE service name. If `server` is a string, that string is used as the DDE service name.

The `topic` argument specifies the DDE topic name to be used in the conversation. If it is a symbol, the symbol’s print name is used. If it is a string, the string is used.

The `class` argument specifies the class of the conversation object to create. It must be a subclass of `dde-client-conversation`, or nil. If it is nil (the default value), then a conversation of class `dde-client-conversation` is created, unless `server` names a client service registered with `define-dde-client`, in which case the registered class (if any) is used.

On executing successfully, this function returns a conversation object. If unsuccessful, the behavior depends on the
value of errorp. If errorp is t (the default value), then an error is raised. If errorp is false, the function returns nil.

Note that conversation objects may only be used within the thread in which they were created.

See also dde-disconnect

### dde-disconnect

**Function**

**Summary**  Disconnects a conversation object.

**Package**  win32

**Signature**  dde-disconnect conversation => result

**Arguments**  conversation   A conversation object.

**Values**  result   A boolean.

**Description**  The function dde-disconnect disconnects the conversation object. The conversation may no longer be used. If the conversation disconnects successfully, t is returned.

See also dde-connect

### dde-execute

**Function**

**Summary**  An alternative syntax for dde-execute-command.

**Package**  win32

**Signature**  dde-execute conversation command &rest {args}* => result

**Arguments**  conversation   A conversation object.
This chapter applies only to LispWorks for Windows

command A string or symbol.
args An argument.

Values result A boolean.

Description The function \texttt{dde-execute} provides an alternative syntax for \texttt{dde-execute-command}. Unlike \texttt{dde-execute-command}, \texttt{dde-execute} takes the arguments for \texttt{command} as a sequence of \texttt{args} following \texttt{&rest}, and does not have an argument for specifying how to handle an error.

See also \texttt{dde-execute*} \texttt{dde-execute-command*} \texttt{dde-execute-string}

\textbf{dde-execute*} \hspace{1cm} \textit{Function}

Summary An alternative syntax for \texttt{dde-execute-command*}.

Package \texttt{win32}

Signature \texttt{dde-execute* service topic command &rest \{args\} => result}

Arguments service A string or symbol.
topic A string symbol.
command A string or symbol.
args An argument.

Values result A boolean.

Description The function \texttt{dde-execute*} provides an alternative syntax for \texttt{dde-execute-command*}. Unlike \texttt{dde-execute-command*}, \texttt{dde-execute*} takes the arguments for \texttt{command} as a
sequence of args following &rest, and does not have any arguments for specifying how to handle errors.

See also  
  dde-execute  
  dde-execute-command  
  dde-execute-string

**dde-execute-command**

*Function*

**Summary**  Sends a command string to a specified conversation.

**Package**  win32

**Signature**  

dde-execute-command conversation command arg-list &key errorp  
=> result

**Arguments**  
  *conversation*  A conversation object.  
  *command*  A string or symbol.  
  *arg-list*  A list of strings, integers, and floats.  
  *errorp*  A boolean.

**Values**  
  *result*  A boolean.

**Description**  The function **dde-execute-command** sends a command string to the conversation specified by *conversation*. The command string consists of *command* and *arg-list*, which are combined using the appropriate argument-marshalling conventions. By default, the syntax is


c[command(arg1, arg2, ...)]

On success, this function returns a result of *t*. On failure, the behavior depends on the value of the *errorp* argument. If *errorp* is *t* (the default value), LispWorks signals an error. If it is *nil*, the function returns *nil* to indicate failure.
See also dde-execute
dde-execute-string

dde-execute-command* Function

Summary Sends a command string to a specified service on a given topic.

Package win32

Signature dde-execute-command* service topic command arg-list &key errorp connect-error-p new-conversation-p => result

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>service</td>
<td>A string or symbol.</td>
</tr>
<tr>
<td>topic</td>
<td>A string or symbol.</td>
</tr>
<tr>
<td>command</td>
<td>A string or symbol.</td>
</tr>
<tr>
<td>arg-list</td>
<td>A list of strings, integers, and floats.</td>
</tr>
<tr>
<td>errorp</td>
<td>A boolean.</td>
</tr>
<tr>
<td>connect-error-p</td>
<td>A boolean.</td>
</tr>
<tr>
<td>new-conversation-p</td>
<td>A boolean.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>A boolean.</td>
</tr>
</tbody>
</table>

Description The function dde-execute-command* is similar to dde-execute-command, and sends a command string to the server specified by service on a topic given by topic. The command string consists of command and arg-list, which are combined using the appropriate argument-marshalling conventions. By default, the syntax is

[command(arg1,arg2,...)]
If `server` names a client service registered with `define-dde-client`, the registered service name is used as the DDE service name. If `server` is any other symbol, the print name of the symbol is used as the DDE service name. If `server` is a string, that string is used as the DDE service name.

The `topic` argument specifies the DDE topic name to be used in the conversation. If it is a symbol, the symbol’s print name is used. If it is a string, the string is used.

If necessary, the function `dde-execute-command*` creates a conversation for the duration of the transaction, but if a suitable conversation already exists, the transaction is executed over that conversation. Hence, if several transactions will be made with the same `service` and `topic`, placing them inside a `with-dde-conversation` prevents a new conversation being established for each transaction.

If `new-conversation-p` is set to `t` a new conversation is always established for the transaction. This new conversation is always automatically disconnected when the transaction is completed.

If `connect-error-p` is `t` (the default value) and a conversation cannot be established, then LispWorks signals an error. If it is `nil`, `dde-execute-command*` returns `nil` if a conversation cannot be established. This allows the caller to distinguish between the cases when the server is not running, and when the server is running but the transaction fails.

Upon success, this function returns a result of `t`. On failure, the behavior depends on the value of the `errorp` argument. If `errorp` is `t` (the default value), LispWorks signals an error. If it is `nil`, the function returns `nil` to indicate failure.

See also
- `dde-execute`
- `dde-execute-string`
- `dde-execute-command`
**dde-execute-string**

**Function**

**Summary**
Issues an execute transaction consisting of a specified string.

**Package**
win32

**Signature**
dde-execute-string conversation command &key errorp => result

**Arguments**
- conversation: A conversation object.
- command: A string or symbol.
- errorp: A boolean.

**Values**
- result: A boolean.

**Description**
The function `dde-execute-string` issues an execute transaction consisting of the string `command`. This string should be appropriately formatted as described in “Execute transactions” on page 323. No processing of the string is performed.

On success, this function returns `t`. On failure, the behavior depends on the value of the `errorp` argument. If `errorp` is `t` (the default value), LispWorks signals an error. If it is `nil`, the function returns `nil` to indicate failure.

**See also**
- dde-execute
- dde-execute-command
- dde-execute-string*

**dde-execute-string***

**Function**

**Summary**
Issues an execute transaction consisting of a specified string on an automatically managed conversation.

**Package**
win32
The DDE client interface

This chapter applies only to LispWorks for Windows

Signature

dde-execute-string* service topic command &key errorp
connect-error-p new-conversation-p => result

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>service</td>
<td>A symbol or string.</td>
</tr>
<tr>
<td>topic</td>
<td>A symbol or string.</td>
</tr>
<tr>
<td>command</td>
<td>A string or symbol.</td>
</tr>
<tr>
<td>errorp</td>
<td>A boolean.</td>
</tr>
<tr>
<td>connect-error-p</td>
<td>A boolean.</td>
</tr>
<tr>
<td>new-conversation-p</td>
<td>A boolean.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>A boolean.</td>
</tr>
</tbody>
</table>

Description

The function **dde-execute-string** is similar to **dde-execute-string**, in that it issues an execute transaction consisting of the string **command**. However, the conversation across which **command** is issued is managed automatically. No processing of the string is performed.

If **server** names a client service registered with **define-dde-client**, the registered service name is used as the DDE service name. If **server** is any other symbol, the print name of the symbol is used as the DDE service name. If **server** is a string, that string is used as the DDE service name.

The **topic** argument specifies the DDE topic name to be used in the conversation. If it is a symbol, the symbol's print name is used. If it is a string, the string is used.

If necessary, the function **dde-execute-string** will create a conversation for the duration of the transaction, but if a suitable conversation already exists, the transaction will be executed over that conversation. Hence, if several transactions will be made with the same **service** and **topic**, placing them inside a **with-dde-conversation** prevents a new conversation being established for each transaction.
If `new-conversation-p` is set to `t` a new conversation is always established for the transaction. This new conversation is always automatically disconnected when the transaction is completed.

If `connect-error-p` is `t` (the default value), then LispWorks signals an error if a conversation cannot be established. If it is `nil`, `dde-execute-string` returns `nil` if a conversation cannot be established. This allows the caller to distinguish between the cases when the server is not running, and when the server is running but the transaction fails.

Upon success, the function returns `t`. On failure, the behavior depends on the value of the `errorp` argument. If `errorp` is `t` (the default value), LispWorks signals an error. If it is `nil`, the function returns `nil` to indicate failure.

See also

`dde-execute`
`dde-execute-command`
`dde-execute-string`

### dde-item

**Accessor**

**Summary**
An accessor which can perform a request transaction or a poke transaction.

**Package**
`win32`

**Signature**
`dde-item conversation item &key format type errorp => result`

**Arguments**
- `conversation` A conversation object.
- `item` A string or symbol.
- `format` A clipboard format specifier.
- `type` A keyword.
- `errorp` A boolean.
Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>A boolean.</td>
</tr>
</tbody>
</table>

Description

The accessor **dde-item** performs a request transaction when read. It performs a poke transaction when set.

To illustrate, the following **dde-request** command

```
(dde-request conversation item :format format :type type :errorp errorp)
```

can also be issued using **dde-item** as follows:

```
(dde-item conversation item :FORMAT format :TYPE type :ERRORP errorp)
```

Similarly, the following **dde-poke** command

```
(dde-poke conversation item data :format format :type type :errorp errorp)
```

can be issued using **dde-item** as follows:

```
(setf (dde-item conversation item :format format :type type :ERRORP errorp) data)
```

except that the **format** always returns **data**.

Upon success, this function returns a **result** of **t**. On failure, the behavior depends on the value of the **errorp** argument. If **errorp** is **t** (the default value), LispWorks signals an error. If it is **nil**, the function returns **nil** to indicate failure.

See also
- **dde-item**
- **dde-poke**
- **dde-request**

### **dde-item**

**Accessor**

**Summary**

An accessor which can perform a request transaction or a poke transaction on an automatically managed conversation.

**Package**

**win32**
This chapter applies only to LispWorks for Windows

**Signature**

```
(dde-item* service topic item &key format type errorp connect-
error-p new-conversation-p => result)
```

**Arguments**

- `service` A string or symbol.
- `topic` A string or symbol.
- `item` A string or symbol.
- `format` A clipboard format specifier.
- `type` A keyword.
- `errorp` A boolean.
- `connect-error-p` A boolean.
- `new-conversation-p` A boolean.

**Values**

- `result` A boolean.

**Description**

The accessor `dde-item*` is similar to `dde-item`, and performs a request transaction when read. It performs a poke transaction when set.

To illustrate, the following `dde-request*` command

```
(dde-request* service topic item :format format :type type :
errorp errorp connect-error-p new-conversation-p)
```

can also be issued using `dde-item*` as follows:

```
(dde-item* service topic item :FORMAT format :TYPE type :
ERRORP errorp connect-error-p new-conversation-p)
```

Similarly, the following `dde-poke*` command

```
(dde-poke* conversation item data :format format :type type :
errorp errorp connect-error-p new-conversation-p)
```

can be issued using `dde-item*` as follows:

```
(setf (dde-item* conversation item :format format :type type :
errorp errorp connect-error-p new-conversation-p) data)
```
except that the format always returns data.

If necessary, the accessor dde-item* creates a conversation for the duration of the transaction, but if a suitable conversation already exists, the transaction is executed over that conversation. If you need to make several transactions with the same service and topic, placing them inside a with-dde-conversation prevents a new conversation being established for each transaction.

If new-conversation-p is set to t a new conversation is always established for the transaction. This new conversation is always automatically disconnected when the transaction is completed.

If connect-error-p is t (the default value), then LispWorks signals an error if a conversation cannot be established. If it is nil, dde-item* returns nil if a conversation cannot be established. This allows the caller to distinguish between the cases when the server is not running, and when the server is running but the transaction fails.

On success, the function returns t. On failure, the behavior depends on the value of the errorp argument. If errorp is t (the default value), LispWorks signals an error. If it is nil, the function returns nil to indicate failure.

See also  
dde-item  
dde-poke  
dde-request

**dde-poke**

*Function*

**Summary**

Issues a poke transaction on a conversation, to set the value of a specified item.

**Package**

win32
This chapter applies only to LispWorks for Windows

Signature

\texttt{dde-poke conversation item data \&key format type errorp => result}

Arguments

- \texttt{conversation} A conversation object.
- \texttt{item} A string or symbol.
- \texttt{data} A string.
- \texttt{format} A clipboard format specifier.
- \texttt{type} A keyword.
- \texttt{errorp} A boolean.

Values

- \texttt{result} A boolean.

Description

The function \texttt{dde-poke} issues a poke transaction on \texttt{conversation} to set the value of the item specified by \texttt{item} to the value specified by \texttt{data}. The argument \texttt{item} should be a string, or a symbol. If it is a symbol its print name is used.

The argument \texttt{format} should be one of the following:

- A DDE format specifier, consisting of either a standard clipboard format or a registered clipboard format.

- A string containing either the name of a standard clipboard format (without the \texttt{CF_} prefix), or the name of a registered clipboard format.

- A symbol, in which case its print name is taken to specify the clipboard format.

- The keyword \texttt{:text}. This is the default value.

The keyword \texttt{:text} is treated specially. If supported by the server it uses the \texttt{CF_UNICODETEXT} clipboard format, otherwise it used the \texttt{CF_TEXT} format.

For text transactions, the default value of \texttt{type} indicates that \texttt{data} is a Lisp string to be used. If \texttt{type} is \texttt{:string-list}, then \texttt{data} is taken to be a list of strings, and is sent as a tab-separated string.
Alternatively, `data` can be a `clipboard-item` structure, containing a foreign pointer to the data to send and the length of the data. In this case the `type` argument is ignored.

On success, this function returns `t`. On failure, the behavior depends on the value of the `errorp` argument. If `errorp` is `t` (the default value), LispWorks signals an error. If it is `nil`, the function returns `nil` to indicate failure.

See also

- `dde-item`
- `dde-request`

### `dde-poke*` Function

**Summary**

Issues a poke transaction on an automatically managed conversation, to set the value of a specified item.

**Package**

`win32`

**Signature**

```
dde-poke* service topic item data &key format type errorp connect-error-p new-conversation-p => result
```

**Arguments**

- `service` A symbol or string.
- `topic` A symbol or string.
- `item` A string or symbol.
- `data` A string.
- `format` A clipboard format specifier.
- `type` A keyword.
- `errorp` A boolean.
- `connect-error-p` A boolean.
- `new-conversation-p` A boolean.
This chapter applies only to LispWorks for Windows

<table>
<thead>
<tr>
<th>Values</th>
<th>result</th>
<th>A boolean.</th>
</tr>
</thead>
</table>

**Description**

The function `dde-poke*` is the same as `dde-poke`, except that conversations are managed automatically. The function issues a poke transaction to set the value of the item specified by `item` to the value specified by `data`. The argument `item` should be a string, or a symbol. If it is a symbol its print name is used.

If `server` names a client service registered with `define-dde-client`, the registered service name is used as the DDE service name. If `server` is any other symbol, the print name of the symbol is used as the DDE service name. If `server` is a string, that string is used as the DDE service name.

The `topic` argument specifies the DDE topic name to be used in the conversation. If it is a symbol, the symbol’s print name is used. If it is a string, the string is used.

For information on the `format`, `type`, and `errorp` arguments, see `dde-poke`.

If necessary, the function `dde-poke*` creates a conversation for the duration of the transaction, but if a suitable conversation already exists, the transaction is executed over that conversation. Hence, if several transactions are made with the same `service` and `topic`, placing them inside a `with-dde-conversation` prevents a new conversation being established for each transaction.

If `new-conversation-p` is set to `t` a new conversation is always established for the transaction. This new conversation is always automatically disconnected when the transaction is completed.

If `connect-error-p` is `t` (the default value), LispWorks signals an error if a conversation cannot be established. If it is `nil`, `dde-poke*` returns `nil` if a conversation cannot be established. This allows the caller to distinguish between the cases when...
the server is not running, and when the server is running but the transaction fails.

See also  
  dde-item  
  dde-request

---

**dde-request**  
*Function*

**Summary**  
Issues a request transaction on a conversation for a specified item.

**Package**  
win32

**Signature**  

dde-request  
  conversation item &key format type errorp => result  
  successp

**Arguments**  
  
  *conversation*  
  A conversation object.

  *item*  
  A string or symbol.

  *format*  
  A clipboard format specifier.

  *type*  
  A keyword.

  *errorp*  
  A boolean.

**Values**  
  
  *result*  
  The return value of the transaction.

  *successp*  
  A boolean.

**Description**  
The function **dde-request** issues a request transaction on **conversation** for the specified **item**. The argument **item** should be a string, or a symbol. If it is a symbol its print name is used.

The argument **format** should be one of the following:

- A DDE format specifier, consisting of either a standard clipboard format or a registered clipboard format.
This chapter applies only to LispWorks for Windows

- A string containing either the name of a standard clipboard format (without the CF_ prefix), or the name of a registered clipboard format.
- A symbol, in which case its print name is taken to specify the clipboard format.
- The keyword :text. This is the default value.

The keyword :text is treated specially. If supported by the server it uses the CF_UNICODETEXT clipboard format, otherwise it used the CF_TEXT format.

The default conversation class only supports text formats, unless type is specified as :foreign. The argument type specifies how the response data should be converted to a Lisp object. For text formats, the default value indicates that a Lisp string should be created. The value :string-list may be specified for type to indicate that the return value should be taken as a tab-separated list of strings; in this case the Lisp return value is a list of strings. The value :foreign can be used with any clipboard format. It returns a clipboard-item structure, containing a foreign pointer to the data, the data length, and the format identifier.

This function returns two values, result and successp. If successful, result is the return value of the transaction (which may be nil in the case of :string-list), and successp is true to indicate success.

On failure, the result of the function depends on the errorp argument. If errorp is t (the default), the function signals an error. If errorp is nil, the function returns (values nil nil).

See also
dde-item
dde-poke
dde-request*
**dde-request**

**Function**

**Summary**
Issues a request transaction on an automatically managed conversation for a specified item.

**Package**
`win32`

**Signature**
```
dde-request* service topic item &key format type errorp connect-error-p new-conversation-p => result successp
```

**Arguments**
- `service` A symbol or string.
- `topic` A symbol or string.
- `item` A string or symbol.
- `format` A clipboard format specifier.
- `type` A keyword.
- `errorp` A boolean.
- `connect-error-p` A boolean.
- `new-conversation-p` A boolean.

**Values**
- `result` The return value of the transaction.

**Description**
The function `dde-request*` is similar to `dde-request`, except that conversations are managed automatically. The function issues a request transaction for the specified `item`. The argument `item` should be a string, or a symbol. If it is a symbol its print name is used.

If `server` names a client service registered with `define-dde-client`, the registered service name is used as the DDE service name. If `server` is any other symbol, the print name of the symbol is used as the DDE service name. If `server` is a string, that string is used as the DDE service name.
This chapter applies only to LispWorks for Windows

The topic argument specifies the DDE topic name to be used in the conversation. If it is a symbol, the symbol’s print name is used. If it is a string, the string is used.

For information on the format, type, and errorp arguments see dde-request.

If necessary, the function dde-request* will create a conversation for the duration of the transaction, but if a suitable conversation already exists, the transaction will be executed over that conversation. Hence, if several transactions will be made with the same service and topic, placing them inside a with-dde-conversation prevents a new conversation being established for each transaction.

If new-conversation-p is set to t a new conversation is always established for the transaction. This new conversation is always automatically disconnected when the transaction is completed.

If connect-error-p is t (the default value), then LispWorks signals an error if a conversation cannot be established. If it is nil, dde-request* returns nil if a conversation cannot be established. This allows the caller to distinguish between the cases when the server is not running, and when the server is running but the transaction fails.

See also dde-item dde-poke dde-request

define-dde-client

Macro

Summary Registers a client service.

Package win32

Signature define-dde-client name &key service class => name
### The DDE client interface

This chapter applies only to LispWorks for Windows

#### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A symbol.</td>
</tr>
<tr>
<td>service</td>
<td>A string.</td>
</tr>
<tr>
<td>class</td>
<td>A subclass of <code>dde-client-conversation</code>.</td>
</tr>
</tbody>
</table>

#### Values

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A symbol.</td>
</tr>
</tbody>
</table>

#### Description

The macro `define-dde-client` defines a mapping from the symbol `name` to the DDE service name with which to establish a conversation, and the conversation class to use for this conversation. The argument `service` is a string which names the DDE service. It defaults to the print-name of `name`. The argument `class` is a subclass of `dde-client-conversation` which is used for all conversations with this service. It defaults to `dde-client-conversation`. Specifying a subclass allows various aspects of the behavior of the conversation to be specialized.

Note that it is generally not necessary to register client services unless a specialized conversation type is required. However, it is sometimes convenient to register a client service in order to allow the service name to be changed in the future.

If the macro executes successfully, the `name` of the DDE service is returned.

#### See also

- `dde-connect`
- `dde-disconnect`
- `with-dde-conversation`

---

### with-dde-conversation

**Macro**

**Summary**

Dynamically binds a conversation to a server across a given body of code.

**Package**

`win32`
This chapter applies only to LispWorks for Windows

Signature
with-dde-conversation (conv service topic &key errorp new-conversation-p) &body body => result

Arguments
conv A conversation object.

service A symbol or string.

topic A symbol or string.

errorp A boolean.

new-conversation-p
A boolean.

body A list of Lisp forms.

Values
result A boolean.

Description
The macro with-dde-conversation dynamically binds a conversation with a server across the scope of a body of code specified by body. The argument conv is bound to a conversation with the server specified by service, and the topic specified by topic.

If server names a client service registered with define-dde-client, the registered service name is used as the DDE service name. If server is any other symbol, the print name of the symbol is used as the DDE service name. If server is a string, that string is used as the DDE service name.

The topic argument specifies the DDE topic name to be used in the conversation. If it is a symbol, the symbol’s print name is used. If it is a string, the string is used.

An existing conversation may be used, if available, unless new-conversation-p is true, in which case a new conversation is always created.

If a new conversation is created, it is disconnected after body has executed as an implicit program.
If a conversation cannot be established, the result returned by the function depends on the value of \texttt{errorp}. If \texttt{errorp} is \texttt{t} (the default value), then LispWorks signals an error. If \texttt{errorp} is \texttt{nil}, the body is not executed, and \texttt{nil} is returned.

See also \texttt{define-dde-client}
This chapter describes the Dynamic Data Exchange (DDE) server interface which is available in the WIN32 package. You should use this chapter in conjunction with Chapter 22, “Dynamic Data Exchange”.

The WIN32 package also includes “Miscellaneous WIN32 symbols”, “The Windows registry API” and “The DDE client interface”. These are documented in separate chapters in this manual.

Note: the WIN32 package is not a supported implementation of the Win32 API. You should not use symbols in the WIN32 package unless they are documented in this manual. Instead, define your own interfaces to Windows functions as you need - see the LispWorks Foreign Language Interface User Guide and Reference Manual for details.

Note: this chapter applies only to LispWorks for Windows, and not the UNIX, Linux, x86/x64 Solaris, FreeBSD, AIX or Mac OS X platforms.

dde-server-poke

Generic Function

Summary

Called when a poke transaction is received.

Package

win32
The DDE server interface

This chapter applies only to LispWorks for Windows

**Signature**

```lisp
dde-server-poke server topic item data &key format &allow-other-keys => successp
```

**Arguments**

- `server` A server object.
- `topic` A topic object.
- `item` A string.
- `data` A string.
- `format` A keyword.

**Values**

- `successp` A boolean.

**Description**

The generic function `dde-server-poke` is called in response to a poke transaction. A method specializing on the classes of `server` and `topic` should poke the data given by `data` into the item specified by `item`.

The keyword `format` indicates the format in which the item is being requested. By default, only text transfers are supported (and the `format` argument will have the value `:text`).

The set of supported formats may be extended in future releases, so applications should always check the value of the format parameter and reject transactions which use formats not supported by the application.

If the poke transaction is successful, non-nil should be returned, and `nil` should be returned for failure.

**See also**

- `dde-poke`
- `dde-request`
- `dde-server-request`

---

**dde-server-request**

*Generic Function*

**Summary**

Called when a request transaction is received.
This chapter applies only to LispWorks for Windows

Package win32

Signature dde-server-request server topic item &key format &allow-other-keys => data

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server</td>
<td>A server object.</td>
</tr>
<tr>
<td>topic</td>
<td>A topic object.</td>
</tr>
<tr>
<td>item</td>
<td>A string.</td>
</tr>
<tr>
<td>format</td>
<td>A keyword.</td>
</tr>
</tbody>
</table>

Values data The returned data.

Description The generic function dde-server-request is called in response to a request transaction. A method specializing on the classes of server and topic should return the data in item.

The expected format of the data is given by format, which defaults to :text. The set of supported formats may be extended in future releases, so applications should always check the value of the format parameter and reject transactions which use formats not supported by the application.

If the request fails, nil should be returned.

See also dde-poke
dde-request
dde-server-poke

dde-server-topic

Generic Function

Summary Called whenever a client attempts to connect to a server with a given topic.

Package win32

Signature dde-server-topic server topic-name => topic
### dde-server-topic

**Arguments**
- `server`: A server.
- `topic-name`: A string.

**Values**
- `topic`: A topic.

**Description**
The generic function `dde-server-topic` is called whenever a client attempts to make a connection to the server. The argument `topic-name` is a string identifying a topic. If the server recognizes the topic, a method specializing on the server should return an instance of one of the server’s topic classes. If the server does not recognize the topic, the method should return `nil`.

**See also**
- `dde-server-topics`
- `dde-topic-items`

---

### dde-server-topics

**Generic Function**

**Summary**
Returns a list of the available general topics on a given server.

**Package**
`win32`

**Signature**
`dde-server-topics server => topic-list`

**Arguments**
- `server`: A server object.

**Values**
- `topic-list`: A list of strings.

**Description**
The generic function `dde-server-topics` returns a list of the available general topics on a given server. A suitable method specializing on the server class should be defined. Dispatching topics (see `define-dde-dispatch-topic`) should not be returned, as they are handled automatically by LispWorks. If you do not provide a `dde-server-topics` method, the default method returns `:unknown`, which prevents the DDE server from responding to the topics request.
Generally only one canonical name should be returned for each topic, even though the server may recognize several alternative forms of name for a topic. For example, if an application implements a topic for each open file, the topics foo, foo.doc and c:\foo.doc may all be acceptable strings for referring to the same topic; however dde-server-topics should return each topic once only.

The application must also provide a method on the dde-server-topic generic function.

See also dde-server-topic
dde-topic-items

dde-system-topic

Class

Summary A built-in topic class for the :system topic.

Package win32

Superclasses dde-topic

Description The class dde-system-topic is a built-in topic class for the :system topic.

See “The system topic” on page 329 for details of the items implemented by this topic.

See also dde-topic

dde-topic

Class

Summary The ancestor of all topic classes.

Package win32
Superclasses  standard-object
Subclasses  dde-system-topic
Description  The class dde-topic is the superclass of all topic objects. You can define subclasses using defclass and return instances of them by defining a method for the dde-server-topic generic function. This allows you to create topics with arbitrary internal state that can be accessed via DDE.
Examples  See examples\dde\server-dispatching.lisp
See also  dde-server-topic
dde-system-topic

dde-topic-items  Generic Function
Summary  Returns the valid items in a topic.
Package  win32
Signature  dde-topic-items server topic => item-strings
Arguments  server  A server object.
topic  A topic object.
Values  item-strings  A list of strings.
Description  The generic function dde-topic-items returns a list of strings corresponding to the valid items in the topic. A method specializing on a server and topic should be defined. If it is not practical to return a list of the items (for example, if the list is potentially infinite), the generic function returns :unknown.
This chapter applies only to LispWorks for Windows

See also  

dde-server-topic  
dde-server-topics

**define-dde-dispatch-topic**  

*Macro*

**Summary**  
Defines a dispatch topic.

**Package**  
win32

**Signature**  

```
define-dde-dispatch-topic name &key server topic-name => name
```

**Arguments**  

- **name**: A symbol.
- **server**: A server class.
- **topic-name**: A string.

**Values**  

- **name**: A symbol.

**Description**  
The macro `define-dde-dispatch-topic` defines a dispatching topic. A dispatching topic is a topic which has a fixed name and always exists. Dispatching topics provide dispatching capabilities, whereby appropriate application-supplied code is executed for each supported transaction. Note that the server implementation also provides some dispatching capabilities.

The name of the dispatching topic object is specified by `name`.

The topic is identified by the string `topic-name`.

The class of the server to attach the topic to is given by `server`.

The macro `define-dde-dispatch-topic` returns the name of the dispatching topic, `name`.

Use `define-dde-server-function` with the :topic option to define items for a dispatch topic.
Example

```lisp
(define-dde-dispatch-topic topic1 :server demo-server)
(define-dde-server-function (item1 :topic topic1) :request ()
  ;; handle topic1.item1 request..)
```

See also
dde-server-topic
dde-server-topics
define-dde-server-function

define-dde-server

**Macro**

Summary

Defines a class for a Lisp DDE server.

Package

**win32**

Signature

```lisp
define-dde-server class-name service-name => class-name
define-dde-server class-name superclasses slot-specs options => class-name
```

Arguments

- **class-name**: A class name.
- **service-name**: A string.
- **superclasses**: A list of superclasses.
- **slot-specs**: The specifications for the class’ slots.
- **options**: A keyword option.

Values

- **class-name**: A class name.

Description

The macro **define-dde-server** defines a class for a Lisp DDE server. The class inherits from **dde-server**.

The long form of the macro is similar to **defclass**, but with one extra option, **:service**, which is used to specify the service name string to which this server will respond.
The short form is provided to handle the common simple case; class-name is the name of the Lisp class to be defined, and service-name is the service name string to which this server will respond.

Example

The first example uses the short version of `define-dde-server` to define a class, called `lisp-server`, which has the service name “LISP”.

```
(define-dde-server lisp-server "LISP")
```

The second example shows how to use the long form of the macro to define the same class, and illustrates the use of the superclasses and options arguments.

```
(define-dde-server lisp-server (dde-server)
 ()
 (:service "LISP"))
```

See also

dde-server-topic
dde-server-topics
dde-topic-items

define-dde-server-function

Macro

Summary

Defines a server function that is called when a specific transaction occurs.

Package

win32

Signature

```
define-dde-server-function name-and-options transaction
(binding*) form* => name

name-and-options ::= name | (name [[option]])

transaction ::= :request | :poke | :execute

option ::= :server server | :topic-class topic-class | :topic topic | :item item | :format format | :command command |
 :result-type result-type | :advisep advisep
```
The DDE server interface

This chapter applies only to LispWorks for Windows

\[
\text{binding} ::= \text{var-binding} \mid \text{execute-arg-binding}
\]

\[
\text{var-binding} ::= (\text{var} :\text{server}) \mid (\text{var} :\text{topic}) \mid (\text{var} :\text{data}\ [\text{data-type}]) \mid (\text{var} :\text{format})
\]

\[
\text{execute-arg-binding} ::= \text{var} \mid (\text{var} \text{ type-spec})
\]

**Arguments**

- **name**: A symbol.
- **transaction**: A keyword.
- **server**: A server object.
- **topic-class**: A topic class.
- **topic**: A symbol naming a dispatch topic.
- **item**: A string.
- **format**: A keyword.
- **command**: A string.
- **result-type**: A data type.
- **advisep**: A boolean.
- **var**: A variable.
- **data-type**: A data type.
- **type-spec**: A data type.
- **form**: A Lisp form.

**Values**

- **name**: A symbol.

**Description**

The macro `define-dde-server-function` is used to define a server function, called `name`, which is called when a specific transaction occurs. The defined function may either be attached to a server class (using the dispatching capabilities built into the server implementation) or to a named dispatch topic.

- To attach the definition to a server, `:server` should be used to specify the server class. `:topic-class` may be used to specify the topic-class for which this definition
This chapter applies only to LispWorks for Windows

should be used. It can be a symbol which names a
\texttt{topic-class}, or \texttt{t} (meaning All topics, this is the default
for execute transactions), or \texttt{system} (The System topic),
or \texttt{non-system} (any topic except the System topic). In
the case of execute transactions only, \texttt{topic-class}
defaults to \texttt{t}; in all other cases, it must be specified. Typically,
execute transactions ignore the topic of the convers-
sation. Alternatively, you may choose to only support
execute transactions in the system topic.

- A server function may instead be attached to a particular
instance of \texttt{dde-dispatch-topic}, previously defined by
\texttt{define-dde-dispatch-topic}. This is the main use of
dispatching topics. In this case \texttt{topic} should be pro-
vided with a symbol that names a dispatching topic. The
function is installed on that topic, and only applies to
that topic.

In the case of a request or poke transaction, \texttt{item} is a string
defining the item name for which this definition should be
invoked. It defaults to the capitalized print-name of \texttt{name},
with hyphens removed.

For request transactions, the \texttt{:format} option is used to spec-
ify the format understood. It defaults to \texttt{:text}. It can be spec-
fied as \texttt{:all}, in which case the \texttt{:format} binding may be
used to determine the actual format requested (see below).

In the case of an execute transaction, \texttt{command} is a string
specifying the name of the command for which this defini-
tion should be invoked. It defaults to the capitalized print-
name of \texttt{name}, with hyphens removed.

The \texttt{execute-arg-bindings} are only used with execute transac-
tions. They specify the arguments expected. \texttt{type-spec} should
be one of \texttt{t, string, number, integer} or \texttt{float}. If not speci-
fied, \texttt{t} is assumed.

The \texttt{var-bindings} may appear anywhere in the binding list,
and in any order. Binding variables to \texttt{:server} and \texttt{:topic} is
useful with all transaction types. A :server binding causes the variable to be bound to the server object, whereas a :topic binding causes the variable to be bound to the topic object. This allows the server and/or the topic to be referred to in the body of the function.

A :format binding can only be used with request and poke transactions, where an option of :format :all has been specified. It causes the variable specified by var to be bound to the format of data requested or supplied. The body of the defined function should fail the transaction if it does not support the requested format.

A :data binding can only be used with poke transactions. It binds a variable to the data to be poked. For text transfers, the data variable is normally bound to a string. However, if datatype is specified as :string-list, the data in the transaction is interpreted as a tab-separated list of strings, and the data variable is bound to a list of strings.

For execute and poke transactions, the body of the defined function is expected to return t for success and nil for failure.

For request transactions, the body of the defined function is normally expected to return a result value, or nil for failure.

The result-type option may only be specified for request transactions. If it is specified as :string-list, then for text requests the body is expected to return a list of strings, which are used to create a tab-separated list to be returned to the client.

Sometimes, it may be necessary to support returning nil to mean the empty list, rather than failure. In this case, the result-type can be specified as (:string-list t). The body is then expected to return two values: a list of strings, and a flag indicating success.

In the case of execute transactions, the command name and arguments are unmarshalled by the default argument
This chapter applies only to LispWorks for Windows

unmarshalling. This is compatible with the default argument unmarshalling described under `dde-execute-command`. The execute string is expected to be of the following syntax:

```
[command1(arg1,arg2,...)] [command2(arg1,arg2,...)] ...
```

Note that multiple commands may be packed into a single execute transaction. However, `dde-execute-command` does not currently generate such strings.

See also
- `dde-execute-command`
- `define-dde-client`
- `define-dde-dispatch-topic`
- `define-dde-server`

**start-dde-server**

*Function*

**Summary**
Creates and starts an instance of a DDE server.

**Package**
`win32`

**Signature**
`start-dde-server name => server`

**Arguments**
- `name` A DDE server class

**Values**
- `server` A server object

**Description**
The function `start-dde-server` creates an instance of a server of the class specified by `name` which then starts accepting transactions. If successful the function returns the server, otherwise `nil` is returned.

You need to call `start-dde-server` in a thread that will process Windows messages. This can either be done by using `capi:execute-with-interface` to run it in the thread of an application’s main window (if there is one) or by running it in a dedicated thread as in the example. DDE callbacks will happen in this thread.
Example

(mp:process-run-function
 "DDE Server"
 ()
 #'(lambda ()
   (win32:start-dde-server 'lispworks-dde-server)
   (loop
     (mp:wait-processing-events
      nil
      :wait-reason "DDE Request Loop"))))

See also define-dde-server
This chapter describes the C functions available in a LispWorks dynamic library, that is a library created by passing *dll-exports* or *dll-added-files* to *save-image* or *deliver*.

For an overview of this functionality with examples of use, see Chapter 14, “LispWorks as a dynamic library”.

**Note:** this chapter applies only to LispWorks on Microsoft Windows, Macintosh, Linux, x86/x64 Solaris and FreeBSD.

**InitLispWorks**

* C function

**Summary** Provides control over the initialization of a LispWorks dynamic library.

**Signature**

On Windows:

```c
int __stdcall InitLispWorks (int MilliTimeOut, void *BaseAddress, size_t ReserveSize)
```

On Linux, Macintosh, x86/x64 Solaris and FreeBSD:

```c
int InitLispWorks (int MilliTimeOut, void *BaseAddress, size_t ReserveSize)
```
The C function `InitLispWorks` allows you to relocate a LispWorks dynamic library if this is necessary, and offers control of the initialization process.

A LispWorks dynamic library is automatically initialized by any call to its exported symbols, so in most cases there is no need to call `InitLispWorks`. It is however necessary when you need to relocate LispWorks or when you need finer control over the initialization process.

For more information about relocating a LispWorks dynamic library, see “Startup relocation” on page 459.

`MilliTimeOut` specifies the time in milliseconds to wait for LispWorks to finish initializing before returning. `InitLispWorks` checks whether the library was initialized and if not initiates initialization. It then waits at most `MilliTimeOut` milliseconds before returning.

`BaseAddress` specifies the base address for relocation. Can be 0.

`ReserveSize` specifies the reserve size for relocation. Can be 0. `BaseAddress` and `ReserveSize` are interpreted as described in “Startup relocation” on page 459.

Non-negative return values indicate success:

- 1: LispWorks was already initialized or in the process of initializing, and finished initializing by the time `InitLispWorks` returned.
- 0: `InitLispWorks` initialized LispWorks and the initialization finished successfully.

Values in the inclusive range [-1, -99] indicate a timeout:

- -1: `InitLispWorks` started initialization and timed out before LispWorks finished mapping itself from the file.
LispWorks already started initialization, and \texttt{InitLispWorks} timed out before LispWorks finished mapping itself from the file.

\texttt{InitLispWorks} started initialization and timed out after LispWorks mapped itself from the file, but before the initialization was complete.

LispWorks already started initialization, and \texttt{InitLispWorks} timed out after LispWorks mapped itself from the file, but before the initialization was complete.

After \texttt{InitLispWorks} times out, the state of LispWorks can be queried by \texttt{LispWorksState}.

Lower values indicate failure, as follows:

\begin{itemize}
  \item [-1000] Failure to start a thread to do the initialization.
  \item [-1401] The file seems to be corrupted.
  \item [-1402] Failure to map into memory.
  \item [-1403] Failure to read the LispWorks header from the file.
  \item [-1406] Bad base address.
\end{itemize}

Additionally, a value \textit{value} in the inclusive range \([-1400, -1001]\) on Linux, Macintosh, FreeBSD and x86/x64 Solaris platforms indicates an error in a system call. Calculate the \texttt{errno} number by \(-1001 - \textit{value}\).

\textbf{Note:} If LispWorks is already initialized or in the process of being initialized, \texttt{InitLispWorks} does not initiate the process of initialization. Therefore the arguments to \texttt{InitLispWorks} have no effect if LispWorks was already initialized when it is called. On Microsoft Windows, the default behavior is to initialize a LispWorks dynamic library automatically during loading, so this needs to be disabled to use
InitLispWorks effectively. Disable automatic initialization of a library as described for deliver and save-image.

Note: Once QuitLispWorks has returned 0, LispWorks can be initialized again. It is possible to quit and restart LispWorks several times, at the same address or at a different address.

Note: On Linux, Macintosh, FreeBSD and x86/x64 Solaris you can create wrappers to the C functions described in this chapter from your application by writing them in C and adding them to the dynamic library using dll-added-files in deliver and save-image. Such wrappers can be used to add calls to InitLispWorks before actually calling into Lisp.

InitLispWorks is defined in each LispWorks dynamic library. For information about creating a LispWorks dynamic library, see deliver and save-image. For an overview of LispWorks as a dynamic library, see Chapter 14, “LispWorks as a dynamic library”.

See also deliver
LispWorksState
save-image
QuitLispWorks

LispWorksDlsym C function

Summary Returns the address of a foreign callable.

Signature On Windows:

void __stdcall *LispWorksDlsym (const char * name)

On Linux, Macintosh, FreeBSD and x86/x64 Solaris:

void *LispWorksDlsym (const char * name)
**Description**  
The C function **LispWorksDlsym** returns the address of a foreign callable name which is defined in Lisp using `fli:define-foreign-callable`.  

`LispWorksDlsym` first checks whether the LispWorks dynamic library finished initializing, and if not uses `InitLispWorks` to initialize it (with `MilliTimeOut` 200). If this fails `LispWorksDlsym` returns NULL. When the LispWorks dynamic library is initialized, `LispWorksDlsym` returns the address of name, or NULL if it is not defined.  

`LispWorksDlsym` is defined in each LispWorks dynamic library. For information about creating a LispWorks dynamic library, see `deliver` and `save-image`. For an overview of LispWorks as a dynamic library, see Chapter 14, “LispWorks as a dynamic library”.

**See also**  
`InitLispWorks`

---

**LispWorksState**  
*C function*

**Summary**  
Returns the state of a LispWorks dynamic library.

**Signature**  
On Windows:

```c
int __stdcall LispWorksState (int MilliTimeOut)
```

On Linux, Macintosh, FreeBSD and x86/x64 Solaris:

```c
int LispWorksState (int MilliTimeOut)
```

**Description**  
The C function **LispWorksState** returns the state of a LispWorks dynamic library.  

`MilliTimeOut` specifies the time to wait in milliseconds if LispWorks is in the process of initialization.  

If LispWorks has not been initialized, or has been quit by `QuitLispWorks`, `LispWorksState` returns -100. Otherwise, it returns the same values as `InitLispWorks`. In particular, if
LispWorks is already properly initialized it returns 1, and if LispWorks is still in the process of initialization it returns -2 or -4. Otherwise it returns a more negative number indicating an error.

**LispWorksState** is defined in each LispWorks dynamic library. For information about creating a LispWorks dynamic library, see `deliver` and `save-image`. For an overview of LispWorks as a dynamic library, see Chapter 14, “LispWorks as a dynamic library”.

**See also**
- `InitLispWorks`
- `QuitLispWorks`

### SimpleInitLispWorks

**C function**

**Summary**
Initializes a LispWorks dynamic library.

**Signature**

On Windows:

```c
int __stdcall SimpleInitLispWorks (void)
```

On Linux, Macintosh, FreeBSD and x86/x64 Solaris:

```c
int SimpleInitLispWorks (void)
```

**Description**

The C function `SimpleInitLispWorks` calls `InitLispWorks(0, 0, 0)` and returns the value of that call.

`SimpleInitLispWorks` is defined in each LispWorks dynamic library. For information about creating a LispWorks dynamic library, see `deliver` and `save-image`. For an overview of LispWorks as a dynamic library, see Chapter 14, “LispWorks as a dynamic library”.

**See also**
- `InitLispWorks`
**QuitLispWorks**

*C function*

**Summary**
Allows a LispWorks dynamic library to be unloaded.

**Signature**
On Windows:

```c
int __stdcall QuitLispWorks(int Force, int MilliTimeOut)
```

On Linux, Macintosh, FreeBSD and x86/x64 Solaris:

```c
int QuitLispWorks(int Force, int MilliTimeOut)
```

**Description**
The C function *QuitLispWorks* allows a LispWorks dynamic library to be unloaded. You should make a LispWorks dynamic library 'quit' by calling *QuitLispWorks* before unloading the library. This call causes LispWorks to cleanup everything it uses, in particular the memory and threads.

In general, *QuitLispWorks* should be called only when the LispWorks dynamic library is idle. That is, when there is no callback into the library that has not returned, and there are no processes that has started by a callback. All callbacks should return, and any processes should be killed before calling *QuitLispWorks*.

*Force* should be 0 or 1. It specifies whether to force quitting even if LispWorks is still executing something.

*MilliTimeOut* specifies how long to wait for LispWorks to complete the cleanup.

If LispWorks is idle, *QuitLispWorks* signals it to quit, and waits *MilliTimeOut* milliseconds for it to finish the cleanup. If LispWorks finished cleanup, *QuitLispWorks* return 0 (SUCCESS). If the cleanup is not finished it returns -2 (TIMEOUT).

If LispWorks is not idle, that is there are still some active callbacks or there are processes that have started by a callback (even if they are inside process-wait), *QuitLispWorks* checks the value of *Force*. If *Force* is 0, *QuitLispWorks* returns -1 (NOT_IDLE). If *Force* is 1, *QuitLispWorks* signals it to quit and behaves as if LispWorks is idle, described above.
QuitLispWorks can be called repeatedly to check whether LispWorks finished the cleanup.

When QuitLispWorks returns NOT_IDLE, it has done nothing, and the LispWorks dynamic library can be used for further callbacks. Once QuitLispWorks returns any other value, callbacks into the dynamic library will result in undefined behavior.

Once QuitLispWorks returns SUCCESS, it is safe to unload the dynamic library. Unloading it before QuitLispWorks returns SUCCESS gives undefined results.

Once QuitLispWorks returns SUCCESS, LispWorks can be initialized again. Calling any exported function (supplied to save-image or deliver in dll-exports) or any of InitLispWorks, SimpleInitLispWorks and LispWorksDlsym will cause LispWorks to initialize again.

Note: On Linux, Macintosh, FreeBSD and x86/x64 Solaris it is possible to add calls to QuitLispWorks at the right places via dll-added-files.

Note: A possible reason for failure to finish the cleanup is that a LispWorks process is stuck inside a foreign call. Dynamic library applications that need to be unloaded should be careful to ensure that they do not get stuck in a foreign function call.

QuitLispWorks is defined in each LispWorks dynamic library. For information about creating a LispWorks dynamic library, see deliver and save-image. For an overview of LispWorks as a dynamic library, see Chapter 14, “LispWorks as a dynamic library”.

See also deliver
dll-quit
save-image
Index

Symbols
- 552
<$debugger command 15
$(dollar) variable 38
$$ variable 38
$$$ variable 38
>$debugger command 15
>SQL operator 363
?: listener command 7
[... ] syntax in Common SQL 359

Numerics
16-bit-string type 1082
8-bit-string type 1081

A
:a debugger command 20
abort restart 12
accepting-handle type 517
accepting-handle-collection function 518
accepting-handle-local-port function 520
accepting-handle-name function 519
accepting-handle-socket function 520
accepting-handle-user-info function 521
accepts-n-syntax function 1527
accept-connections-creating-async-IO-states function 412, 416, 422, 513
accessor-method-slot-definition generic function 250
accessors
code-coverage-data-create-time 870
code-coverage-data-name 870
code-coverage-file-stats-called 879
code-coverage-file-stats-counters-count 879
code-coverage-file-stats-counters-executed 879
code-coverage-file-stats-counters-hidden 879
code-coverage-file-stats-fully-covered 879
code-coverage-file-stats-hidden-covered 879
code-coverage-file-stats-lambdas-count 879
code-coverage-file-stats-not-called 879
code-coverage-file-stats-partially-covered 879
code-coverage-file-stats-source-file 879
dde-item 1957
dde-item* 1958
df-handle-directory-p 930
df-handle-directory-string 930
df-handle-last-access 930
df-handle-last-modify 930
df-handle-link-p 930
df-handle-size 930
df-handle-writable-p 930
field-access-exception-set-

1993
Index

p 1236
field-exception-class-name 1237
field-exception-field-name 1237
java-array-error-array 1263
java-array-error-caller 1263
java-array-indices-error-indices 1264
java-array-indices-error-rank 1264
java-bad-jobject-caller 1265
java-bad-jobject-object 1265
java-definition-error-class-name 1266
java-definition-error-name 1266
java-exception-exception-name 1268
java-exception-java-backtrace 1268
java-exception-string 1268
java-field-error-field-name 1266
java-field-error-static-p 1266
java-field-error-setting-error-class-name-for-setting 1269
java-field-error-setting-error-field-name 1269
java-field-error-setting-error-new-value 1269
java-instance-jobject 219, 1316
java-method-error-args-num 1266
java-method-error-method-name 1266
java-method-exception-args 1272
java-method-exception-class-name 1272
java-method-exception-method-name 1272
java-method-exception-name 1272
lob-stream-lob-locator 1578
socket-stream-socket 613
sql-database-data-error 1644
sql-error-database-message 1644
sql-error-secondary-error-id 1644
storage-exhausted-gen-num 1885
storage-exhausted-size 1885
storage-exhausted-static 1885
storage-exhausted-type 1885
stream-read-timeout 613
stream-write-timeout 613
action lists 95
defining 96
eamples 97
*active-finders* variable 791
add-code-coverage-data function 839
adding actions to action lists 96
addMessage Java method 1352
add-method generic function 251
address space
in 32-bit LispWorks 458, 460
in 64-bit LispWorks 461, 477
add-special-free-action function 164, 841
add-sql-stream function 372, 1528
add-symbol-profiler function 842
adjust-array function 266
advice
after 65
around 65
before 65
example of use 70
facility 63
for macros 67
for methods 67
main chapter 63, 75
removing 66
after advice 65
:after keyword 50
:all debugger command 18
:all keyword 309
allocated-in-its-own-segment-p function 1715
:allocation keyword 55
allocation of stacks 164, 1738
allocation-in-gen-num macro 137, 146, 843
allowing-block-interrupts macro 289, 1359
analysing-special-variables-
usage macro 844
Android interface
Java side 1339
Lisp side 223, 224
overview 223
Android runtimes
creating 224
element 227
android-build-value function 848
android-funcall-in-main-
thread function 849
android-funcall-in-main-
thread-list function 849
android-get-current-activity
function 851
android-main-thread-p
function 852
ANSI
Common Lisp 252, 351
SQL mode 334, 375
ANSI Common Lisp Standard
menu command 639
ANSI_QUOTES
SQL mode 375
any SQL operator 363
any-capi-window-displayed-p
function 853
any-other-process-non-inte-
nal-server-p function 1361
append macro 1083
append-file function 1082
apply-in-pane-process
function 267
apply-in-wait-state-collec-
tion-process function 411,
522
apply-with-allocation-in-
gen-num function 147, 1716
approaching-memory-limit
condition 1717
apropos function 639
apropos-list function 640
argument list 1137
arguments
command line 454
lisp function 1137
arguments for traced functions 50
around advice 65
array-dimension-limit
constant 479
array-single-thread-p
function 854
array-total-size-limit
constant 479
array-weak-p function 854
ASCII 444, 447
:ascii external format 444
ASDF 306, 311
ASDF2 306, 311
async-io-state structure type 412,
523
async-io-state-abort
function 413, 417, 523
async-io-state-abort-and-
close function 417, 525
async-io-state-address
function 526
async-io-state-attach-ssl
function 422, 425, 526
async-io-state-buffered-
data-length function 414,
528
async-io-state-collection
function 530
async-io-state-ctx function 431,
529
async-io-state-detach-ssl
function 425, 529
async-io-state-discard
function 413, 417, 531
async-io-state-finish
function 413, 417, 532
async-io-state-get-buffered-
data function 414, 417, 533
async-io-state-handshake
function 534
async-io-state-max-read
function 535
async-io-state-name function 536
async-io-state-object
function 536
async-io-state-old-length
function 537
async-io-state-peer-address
function 538
async-io-state-read-buffer
function 413, 416, 538
async-io-state-read-status
function 540
async-io-state-read-timeout
function 541
async-io-state-read-with-
checking function 413, 416,
Index

async-io-state-receive-message function 413, 416, 545
async-io-state-send-message function 413, 416, 547
async-io-state-send-message-to-address function 413, 416, 549
async-io-state-ssl function 431, 551
async-io-state-ssl-side function 431, 551
async-io-state-user-info function 412, 553
async-io-state-write-buffer function 413, 416, 552
async-io-state-write-status function 540
async-io-state-write-timeout function 541
at-location macro 792
atomic-decf macro 1718
atomic-exchange macro 1719
atomic-fixnum-decf macro 1719
atomic-fixnum-incf macro 1719
atomic-incf macro 1718
atomicity and thread-safety in CAPI 267
in the editor 266
in the LispWorks implementation 265
atomic-pop macro 1720
atomic-push macro 1721
attach-ssl function 420, 422, 554
attribute-type function 353, 1529
augmented-string type 1722
augmented-string-p function 1722
augment-environment function 855
*autolload-asdf-integration* variable 1083
avoid-gc function 146, 857
barrier-arriver-count function 1362
barrier-block-and-wait function 1363
barrier-change-count function 1366
barrier-count function 1367
barrier-disable function 287, 1368
barrier-enable function 287, 1368
barrier-name function 1369
barrier-pass-through function 1370
barrier-unblock function 1370
barrier-wait function 287, 1372
base slot 354
base-char type 436, 1084
base-character type 1084
base-character-p function 1084
base-char-code-limit constant 1085
base-char-p function 1085
base-char-ref function 1821
base-string type 438, 641
base-string-p function 1086
:base-table class option 1554
before advice 65
:before keyword 49
binary files 101
*binary-file-type* variable 649, 1723
*binary-file-types* variable 649, 1723
/bindings keyword 757
binds-who function 860
block-promotion macro 861
.bmp external format 444, 827
.bmp-char type 436, 1086
.bmp-char-p function 1087
.bmp-native external format 446, 827
.bmp-reversed external format 446, 827
.bmp-string type 438, 1088
.bmp-string-p function 1089
BOM 448, 450
Bordeaux threads
APIs needed for 1447
:bq debugger command 15
:break keyword 51
break-new-instances-on-access function 485
break-on-access function 486
:break-on-exit keyword 51
break-on-unresolved-functions function 1075
browser 1089
*browser-location* variable 1089
browsing documentation 1089
buffered-stream class 1681
:bug-form listener command 7
BugFormLogsList java class 1357
BugFormViewer java class 1357
building-universal-intermediate-p function 862
byte order
  big-endian architectures 1069
  little-endian architectures 1069
Byte Order Mark 448, 450

C
  :c debugger command 20
C functions
dlopen 191
dlsym 191
GetProcAddress 191
InitLispWorks 194, 460, 1985
LispWorksDlsym 1988
LispWorksState 194, 1989
LoadLibrary 191
QuitLispWorks 195, 1991
SimpleInitLispWorks 1990
cache-table-queries function 349, 1351
*cache-table-queries-default* variable 1532
call Unix functions from Lisp 462
callDoubleAJava method 1332
callDoubleVJava method 1332
calling AppleScript 1725
callIntAJava method 1332
callIntVJava method 1332
call-java-method function 1209
call-java-method-error condition 1210
call-next-advice function 66, 73, 1090
callObjectAJava method 1332
callObjectVJava method 1332
calls-who function 863
call-system function 1724
call-system-showing-output function 1727
callVoidAJava method 1332
callVoidVJava method 1332
call-wait-state-collection function 411, 413, 556
canonicalize-dspec function 793
canonicalize-sid-string function 1895
catch frame, examining 12
:catchers keyword 760
catching-exceptions-bind macro 1211
catching-java-exceptions macro 1211
:caused-by keyword 309
:cc debugger command 18
cd macro 864
cdr-assoc function 1730
change-directory function 187, 864
change-process-priority function 295, 1375
class options
  parsing of 495
  :base-table 1554
  :extra-initargs 487, 489, 658
  :optimize-slot-access 250, 253, 504, 658
classes
  buffered-stream 1681
dde-system-topic 1975
dde-topic 1975
eql-specializer 250
funcallable-standard-object 494
fundamental-binary-input-stream 1683
fundamental-binary-output-stream 1684
fundamental-binary-stream 1684
fundamental-character-
Index

input-stream 1685
fundamental-character-output-stream 1686
fundamental-character-stream 1687
fundamental-input-stream 1687
fundamental-output-stream 1688
fundamental-stream 1688
lob-stream 380, 384, 389, 1577
method-combination 251
socket-error 612
socket-stream 418, 422, 612
ssl-closed 431, 623
ssl-condition 431, 623
ssl-error 431, 624
ssl-failure 431, 624
ssl-x509-lookup 431, 626
standard-accessor-method 250
standard-db-object 354, 1666
standard-java-object 1315
standard-reader-method 250
standard-writer-method 250
storage-exhausted 1884
wait-state-collection 411, 636
class-extra-initargs 487
clean-down function 136, 163, 866
clean-generation-0 function 146, 868
clearBugFormLogs Java method 1350
clear-code-coverage function 869
client-remote-debugging system class 765
CLOS Metaobject Protocol menu command 249
close generic function 641
close-accepting-handle function 557
close-async-io-state function 414, 415, 417, 558
close-registry-key function 468, 1923
close-remote-debugging-connection function 36, 729
close-serial-port function 1519
close-wait-state-collection function 411, 559
:close-initarg-checking delivery keyword 501
Cocoa application 1397
Cocoa application bundle saving 179
Cocoa event loop 1397
Code Coverage
HTML output 875
HTML output index file 876
Source files HTML coloring 877
code coverage 119
Code Coverage Current Buffer Editor command 883
Code Coverage File Editor command 883
code signing in saved image 1006
code-coverage-data type 870
code-coverage-data-create-time accessor 870
code-coverage-data-generate-coloring-html function 871
code-coverage-data-generate-statistics function 877
code-coverage-data-name accessor 870
code-coverage-file-stats type 879
code-coverage-file-stats-called accessor 879
code-coverage-file-stats-counters-count accessor 879
code-coverage-file-stats-counters-executed accessor 879
code-coverage-file-stats-counters-hidden accessor 879
code-coverage-file-stats-fully-covered accessor 879
code-coverage-file-stats-hidden-covered accessor 879
code-coverage-file-stats-lambdas-count accessor 879
code-coverage-file-stats-not-called accessor 879
code-coverage-file-stats-partially-covered accessor 879
code-coverage-file-stats-source-file accessor 879
code-coverage-set-editor-colors function 881
Index

code-coverage-set-editor-
default-data function 883
code-coverage-set-html-background-colors function 884
code-page external format 443
coerce function 642
coerce-to-gesture-spec function 1731
*coff-loading-verbose* 1080
collect-generation-2 function 144, 146, 885
collect-highest-generation function 146, 886
collect-registry-subkeys function 468, 1924
collect-registry-values function 468, 1925
command line 454
command line arguments 1795
-build 454
-display 455
-env 455
-environment 455
-eval 455
-IIOPhost 455
-IIOPnumeric 456
-init 456
-load 456
-lw-no-redirection 456
-multiprocessing 456
-no-restart-function 456
-ORBport 457
--relocate-image 457
--reserve-size 457
-siteinit 457
command line processing 1795
commands
listener 1740
top level 1740
commit function 348, 350, 382, 1532
Common Lisp systems. See system
Common SQL
[... ] syntax 359
case of names 374
database connection 338
database encoding 375
date fields 370, 378
encoding 339
ersors 372
Functional DDL 353
Functional DML 346
functional interface 346
I/O recording 372
intialization 334
iteration 358
main chapter 331
Object Oriented DDL 355
Object Oriented DML 357
object-oriented interface 354
ODBC compliance 333
programmatic interface 368
result types 347, 377
supported databases 333
symbolic syntax 358
transaction handling 339, 350, 377
utilities 370
Common SQL errors
sql-connection-error 373
sql-database-data-error 372
sql-database-error 372
sql-fatal-error 373
sql-temporary-error 372
sql-timeout-error 373
sql-user-error 372
compare-and-swap macro 1733
compilation-speed 105
Compile Buffer Editor command 120
Compile File Editor command 120
compile function 643
compile-file function 645
compile-file-if-needed function 887
compiler
comparison with interpreter 101
control 104
levels of safety 105
main chapter 101
optimization of 104–108
workings of 103
compiler explanations 111, 651
compiler help 111, 651
*compiler-break-on-error* variable 887
compile-system function 306, 1092
compiling
arbitrary forms 103
functions 102
source files 103
compute-applicable-methods-using-classes generic function 251
compute-class-potential-initargs generic function 488
Index

compute-discriminating-function generic function 251, 490
compute-effective-method-function-from-classes
generic function 490
concatenate function 650
concatenate-system function 1094
Condition variables 285
Conditionalization
  LispWorks architectures 675
  LispWorks implementations 672
  LispWorks versions 674
Conditionalization for LispWorks
  versions 674
Conditionalization for the LispWorks architectures 675
Conditionalization for the LispWorks implementations 672
conditions
  approaching-memory-limit 1717
  call-java-method-error 1210
  create-java-object-error 1217
  external-format-error 835
  fasl-error 929
  field-access-exception 1236
  field-exception 1237
  file-encoding-resolution-error 1751
  java-array-error 1263
  java-array-indices-error 1263
  java-array-simple-error 1265
  java-bad-jobject 1265
  java-class-error 1266
  java-definition-error 1266
  java-exception 1267
  java-field-error 1266
  java-field-setting-error 1268
  java-id-exception 1269
  java-instance-without-jobject-error 1270
  java-interface-error 1270
  java-low-level-exception 1271
  java-method-error 1266
  java-method-exception 1271
  java-normal-exception 1272
  java-not-a-java-object-error 1273
  java-not-an-array-error 1274
  java-out-of-bounds-error 1276
  java-serious-exception 1278
  java-storing-wrong-type-error 1276
  sql-connection-error 1643
  sql-database-data-error 1643
  sql-database-error 1643
  sql-fatal-error 1648
  sql-temporary-error 1665
  sql-timeout-error 1665
  sql-user-error 395, 396, 1653, 1659, 1665
condition-variable system
class 1376
condition-variable-broadcast
  function 286, 1376
condition-variable-signal
  function 286, 1377
condition-variable-wait
  function 285, 1378
condition-variable-wait-count
  function 286, 1379
configure-remote-debugging-spec
  function 26, 27, 33, 730
configuring the printer 189
connect
  function 334, 338, 339, 374, 1533
  Connect Remote Debugging Editor command 747
connected-databases function 338, 1542
*connect-if-exists* variable 1541
connecting to a database
  MySQL 339
  ODBC 339
  Oracle 338
  PostgreSQL 342
connect-to-named-pipe
  function 1896
connect-to-tcp-server
  function 572
console 1002
console application 1002
constants
  array-dimension-limit 479
  array-total-size-limit 479
  base-char-code-limit 1085
  *java-null* 1274
  most-positive-fixnum 479
continue restart 12
copy-code-coverage-data
Index

create-view function 354, 1545
create-view-from-class function 355, 1546
creation of process 262
cross-referencing 860, 863, 1040, 1059, 1060:ctx-configure-callback
  initarg 613
current frame 15
current process 262
current-function-name function 897
current-pathname function 1099
*current-process* variable 262,
  1380
current-process-block-interrupts function 289, 1380
current-process-in-cleanup-p function 1381
current-process-kill function 1382
current-process-pause function 1383
current-process-send function 1385
current-process-set-terminate-method function 1386
current-process-unblock-interrupts function 289, 1387
current-stack-length function 898
customization
  main chapter 178
  of editor 185
:cv inspector command 39

D
:d inspector command 39
database
  connection in Common SQL 338
  encoding in Common SQL 375
table names 374
database-name function 338, 1546
databases
  supported 333
dates
  in Common SQL 370, 378
date-string function 899
dde-advice-client-data generic
  function 324
dde-advice-start function 324, 1941
dde-advice-start* function 324,
Index

1944

dde-advise-stop function 325, 1945
dde-advise-stop* function 325, 1947
dde-client-advise-data generic
  function 1948
dde-connect function 323, 1949
dde-disconnect function 323, 1950
dde-execute function 1950
dde-execute* function 1951
dde-execute-command function 326, 1952
dde-execute-command*
  function 326, 1953
dde-execute-string function 326, 1955
dde-execute-string* function 326, 1955
dde-item accessor 1957
dde-item* accessor 1958
dde-item function 326
dde-poke function 325, 1960
dde-poke* function 1962
dde-request function 325, 1964
dde-request* function 1966
dde-server-poke generic
  function 327, 1971
dde-server-request generic
  function 327, 1972
dde-server-topic generic
  function 1973
dde-server-topics generic
  function 328, 1974
dde-system-topic class 1975
dde-topic class 1975
dde-topic-items generic
  function 1976
DDL 353, 355
debug 105
debugger
  break gesture 10
  commands 13
  commands not recognized 23
  control variables 23
  invoking from the tracer 51
  keyboard interrupt 10
  main chapter 9
  remote 25
  troubleshooting 23
debugger commands
  :< 15
  :: 15
  :a 20
  :all 18
  :b 14
  :bg 15
  :c 20
  :cc 18
  :ed 18
  :error 18
  :func 19
  :l 17
  :lambda 19
  :lf 19
  :n 16
  :p 15
  :res 21
  :ret 21
  :top 21
  :v 16
*debug-initialization-
  errors-in-snap-shot* variable 1736
*debug-io* variable 23, 705
debug-other-process
  function 1388
*debug-print-length* variable 23, 738
*debug-print-level* variable 24, 738
declaim macro 110, 650
declaration
  alias 651
  :explain 651
  invisible-frame 651
  lambda-list 651
  special-dynamic 113, 651
  special-fast-access 113, 651
  special-global 113, 651
  values 651
declaration-information
  function 900
declare :explain 111, 651
declare special form 104, 109, 651
decode-external-string
  function 834
decode-to-db-standard-date
  function 1547
decode-to-db-standard-times-
  tamp function 1547
def macro 794
defadvice macro 67, 73, 1100
default directory 952
default file directory 952
*default-action-list-sort-
<table>
<thead>
<tr>
<th>term</th>
<th>page numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>time*</td>
<td>variable 1103, 97</td>
</tr>
<tr>
<td><em>default-character-element-type</em></td>
<td>parameter 438, 440, 441, 442, 449, 1104</td>
</tr>
<tr>
<td><em>default-client-remote-debugging-server-port</em></td>
<td>variable 27, 739</td>
</tr>
<tr>
<td>default-constructor-arguments</td>
<td>generic function 1217</td>
</tr>
<tr>
<td><em>default-database</em></td>
<td>variable 1549</td>
</tr>
<tr>
<td><em>default-database-type</em></td>
<td>variable 1550</td>
</tr>
<tr>
<td>default-eol-style</td>
<td>function 1737</td>
</tr>
<tr>
<td><em>default-ide-remote-debugging-server-port</em></td>
<td>variable 27, 740</td>
</tr>
<tr>
<td><em>default-libraries</em></td>
<td>variable 1079</td>
</tr>
<tr>
<td>default-name-constructor</td>
<td>function 1218</td>
</tr>
<tr>
<td><em>default-package-use-list</em></td>
<td>variable 901</td>
</tr>
<tr>
<td>:default-pathname</td>
<td>keyword 307</td>
</tr>
<tr>
<td><em>default-process-priority</em></td>
<td>variable 1389</td>
</tr>
<tr>
<td><em>default-profiler-collapse</em></td>
<td>variable 901</td>
</tr>
<tr>
<td><em>default-profiler-cutoff</em></td>
<td>variable 902</td>
</tr>
<tr>
<td><em>default-profiler-limit</em></td>
<td>variable 902</td>
</tr>
<tr>
<td><em>default-profiler-sort</em></td>
<td>variable 903</td>
</tr>
<tr>
<td><em>default-stack-group-list-length</em></td>
<td>variable 164, 1738</td>
</tr>
<tr>
<td><em>default-update-objects-max-len</em></td>
<td>variable 1550</td>
</tr>
<tr>
<td>defclass</td>
<td>macro 657</td>
</tr>
<tr>
<td>defglobal-parameter</td>
<td>macro 903</td>
</tr>
<tr>
<td>defglobal-variable</td>
<td>macro 904</td>
</tr>
<tr>
<td>define-action</td>
<td>macro 96, 1105</td>
</tr>
<tr>
<td>define-action-list</td>
<td>macro 96, 1107</td>
</tr>
<tr>
<td>define-atomic-modify-macro</td>
<td>macro 1738</td>
</tr>
<tr>
<td>define-dde-client</td>
<td>function 325</td>
</tr>
<tr>
<td>define-dde-client-macro</td>
<td>macro 1967</td>
</tr>
<tr>
<td>define-dde-dispatch-topic</td>
<td>macro 329, 1977</td>
</tr>
<tr>
<td>define-dde-server</td>
<td>macro 327, 1978</td>
</tr>
<tr>
<td>define-dde-server-function</td>
<td>macro 327, 1979</td>
</tr>
<tr>
<td>define-declaration</td>
<td>macro 905</td>
</tr>
<tr>
<td>define-dspec-alias</td>
<td>macro 795</td>
</tr>
<tr>
<td>define-dspec-class</td>
<td>macro 796</td>
</tr>
<tr>
<td>define-field-accessor</td>
<td>macro 1219</td>
</tr>
<tr>
<td>define-foreign-callable</td>
<td>macro 297</td>
</tr>
<tr>
<td>define-form-parser</td>
<td>macro 90, 799</td>
</tr>
<tr>
<td>define-java-caller</td>
<td>macro 1220</td>
</tr>
<tr>
<td>define-java-callers</td>
<td>macro 1223</td>
</tr>
<tr>
<td>define-java-put-field-accessor</td>
<td>macro 1220</td>
</tr>
<tr>
<td>define-lisp-proxy</td>
<td>macro 1225</td>
</tr>
<tr>
<td>define-top-loop-command</td>
<td>macro 1740</td>
</tr>
<tr>
<td>definition specs</td>
<td>58</td>
</tr>
<tr>
<td>defpackage</td>
<td>macro 661</td>
</tr>
<tr>
<td>defparameter</td>
<td>macro 904</td>
</tr>
<tr>
<td>defparser</td>
<td>error handling with 316</td>
</tr>
<tr>
<td>defparser macro 1515</td>
<td>functions defined by 315</td>
</tr>
<tr>
<td>defrule</td>
<td>compiling a rule dynamically 103</td>
</tr>
<tr>
<td>defstruct</td>
<td>macro 189</td>
</tr>
<tr>
<td><em>defstruct-generates-print-object-method</em></td>
<td>variable 189</td>
</tr>
<tr>
<td>defsystem</td>
<td>macro 305–311, 1109</td>
</tr>
<tr>
<td>examples of use 310</td>
<td></td>
</tr>
<tr>
<td><em>defsystem-verbose</em></td>
<td>variable 1115</td>
</tr>
<tr>
<td>deftransform</td>
<td>macro 725</td>
</tr>
<tr>
<td>defvar</td>
<td>macro 904</td>
</tr>
<tr>
<td>def-view-class</td>
<td>macro 332, 354, 355, 1551</td>
</tr>
<tr>
<td>delete-advice</td>
<td>macro 66, 73, 907</td>
</tr>
<tr>
<td>delete-directory</td>
<td>function 1115</td>
</tr>
<tr>
<td>delete-duplicates</td>
<td>function 116</td>
</tr>
<tr>
<td>delete-instance-records</td>
<td>generic function 357, 1558, 1572</td>
</tr>
<tr>
<td>delete-records</td>
<td>function 348, 349, 1559</td>
</tr>
<tr>
<td>delete-registry-key</td>
<td>function 468, 1929</td>
</tr>
<tr>
<td>delete-sql-stream</td>
<td>function 372, 1560</td>
</tr>
<tr>
<td>deliver</td>
<td>function 191, 454, 460, 1116</td>
</tr>
<tr>
<td>deliverable</td>
<td>filename 454, 1149</td>
</tr>
<tr>
<td>pathname 454, 1149</td>
<td></td>
</tr>
<tr>
<td>delivered-image-p</td>
<td>function 908</td>
</tr>
<tr>
<td>delivering a DLL</td>
<td>191</td>
</tr>
<tr>
<td>delivering a dynamic library</td>
<td>191</td>
</tr>
<tr>
<td>deliver-to-android-project</td>
<td></td>
</tr>
</tbody>
</table>
### Index

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>close-initarg-checking</td>
<td>501</td>
</tr>
<tr>
<td>multiprocessing</td>
<td>264</td>
</tr>
<tr>
<td>quit-when-no-windows</td>
<td>1176</td>
</tr>
<tr>
<td>startup-bitmap-file</td>
<td>1898</td>
</tr>
<tr>
<td>describe</td>
<td>37, 663</td>
</tr>
<tr>
<td><em>describe-length</em></td>
<td>variable 1117, 39</td>
</tr>
<tr>
<td><em>describe-level</em></td>
<td>variable 1117, 38</td>
</tr>
<tr>
<td>describe-object</td>
<td>generic function 38</td>
</tr>
<tr>
<td><em>describe-print-length</em></td>
<td>variable 1119, 38</td>
</tr>
<tr>
<td><em>describe-print-level</em></td>
<td>variable 1119, 38</td>
</tr>
<tr>
<td>destroy-prepared-statement</td>
<td>function 1561</td>
</tr>
<tr>
<td>destroy-ssl</td>
<td>function 431, 574</td>
</tr>
<tr>
<td>destructive-add-code-coverage-data</td>
<td>function 839</td>
</tr>
<tr>
<td>destructive-reverse-subtract-code-coverage-data</td>
<td>function 839</td>
</tr>
<tr>
<td>detach-ssl</td>
<td>function 425, 575</td>
</tr>
<tr>
<td>detect-eol-style</td>
<td>function 1742</td>
</tr>
<tr>
<td>detect-japanese-encoding-in-file</td>
<td>function 1743</td>
</tr>
<tr>
<td>detect-unicode-bom</td>
<td>function 1744</td>
</tr>
<tr>
<td>detect-utf32-bom</td>
<td>function 1744</td>
</tr>
<tr>
<td>detect-utf8-bom</td>
<td>function 1744</td>
</tr>
<tr>
<td>directory</td>
<td>function 663</td>
</tr>
<tr>
<td><em>directory-link-transparency</em></td>
<td>variable 664, 1745</td>
</tr>
<tr>
<td>disable-sql-reader-syntax</td>
<td>function 370, 1561</td>
</tr>
<tr>
<td><em>disable-trace</em></td>
<td>variable 914</td>
</tr>
<tr>
<td>disassemble</td>
<td>function 668</td>
</tr>
<tr>
<td>discard-source-info</td>
<td>function 803</td>
</tr>
<tr>
<td>disconnect</td>
<td>function 338, 1562</td>
</tr>
<tr>
<td>dismiss-splash-screen</td>
<td>function 1898</td>
</tr>
<tr>
<td>DLL</td>
<td>191</td>
</tr>
<tr>
<td>filename</td>
<td>454, 1149</td>
</tr>
<tr>
<td>pathname</td>
<td>454, 1149</td>
</tr>
<tr>
<td>dll-quit</td>
<td>function 195, 1120</td>
</tr>
<tr>
<td>dlopen</td>
<td>C function 191</td>
</tr>
<tr>
<td>dl_syrn</td>
<td>C function 191</td>
</tr>
<tr>
<td><em>dm</em></td>
<td>inspector command 39</td>
</tr>
<tr>
<td>DML</td>
<td>346, 357</td>
</tr>
<tr>
<td>DNS</td>
<td>578</td>
</tr>
<tr>
<td>documentation</td>
<td>generic function 668</td>
</tr>
<tr>
<td>$(dollar)</td>
<td>variable 38</td>
</tr>
<tr>
<td>$$</td>
<td>variable 38</td>
</tr>
<tr>
<td>$$$</td>
<td>variable 38</td>
</tr>
<tr>
<td>domain</td>
<td>578</td>
</tr>
<tr>
<td>do-nothing</td>
<td>function 1123</td>
</tr>
<tr>
<td>:dont-know</td>
<td>keyword 17</td>
</tr>
<tr>
<td>do-profiling</td>
<td>function 169, 915</td>
</tr>
<tr>
<td>do-query</td>
<td>macro 351, 381, 1563</td>
</tr>
<tr>
<td>do-rand-seed</td>
<td>function 576</td>
</tr>
<tr>
<td>DOS command</td>
<td>call-system 1724</td>
</tr>
<tr>
<td>call-system-showing-output</td>
<td>1727</td>
</tr>
<tr>
<td>open-pipe</td>
<td>1823</td>
</tr>
<tr>
<td>DOS window</td>
<td>controlling, in call-system 1725, 1726</td>
</tr>
<tr>
<td>dotted-list-length</td>
<td>function 1122</td>
</tr>
<tr>
<td>dotted-list-p</td>
<td>function 1123</td>
</tr>
<tr>
<td>double-float</td>
<td>type 669</td>
</tr>
<tr>
<td>:dr</td>
<td>inspector command 39</td>
</tr>
<tr>
<td>drop-index</td>
<td>function 354, 1564</td>
</tr>
<tr>
<td>drop-table</td>
<td>function 354, 1565</td>
</tr>
<tr>
<td>drop-view</td>
<td>function 354, 1565</td>
</tr>
<tr>
<td>drop-view-from-class</td>
<td>function 355, 1566</td>
</tr>
<tr>
<td>dspec-class</td>
<td>function 803</td>
</tr>
<tr>
<td><em>dspec-classes</em></td>
<td>variable 804</td>
</tr>
<tr>
<td>dspec-defined-p</td>
<td>function 804</td>
</tr>
<tr>
<td>dspec-definition-locations</td>
<td>function 805</td>
</tr>
<tr>
<td>dspec-equal</td>
<td>function 806</td>
</tr>
<tr>
<td>dspec-name</td>
<td>function 806</td>
</tr>
<tr>
<td>dspec-primary-name</td>
<td>function 807</td>
</tr>
<tr>
<td>dspec-progenitor</td>
<td>function 808</td>
</tr>
<tr>
<td>dspecs</td>
<td>aggregate 82</td>
</tr>
<tr>
<td>canonical</td>
<td>76</td>
</tr>
<tr>
<td>displaying definitions</td>
<td>89</td>
</tr>
<tr>
<td>examples</td>
<td>75</td>
</tr>
<tr>
<td>finding definitions</td>
<td>89</td>
</tr>
<tr>
<td>for subfunctions</td>
<td>85</td>
</tr>
<tr>
<td>grouping definitions</td>
<td>82</td>
</tr>
<tr>
<td>new defining forms</td>
<td>80</td>
</tr>
<tr>
<td>parts</td>
<td>82</td>
</tr>
<tr>
<td>recording definitions</td>
<td>87</td>
</tr>
<tr>
<td>dspec-subclass-p</td>
<td>function 808</td>
</tr>
</tbody>
</table>
dspec-undefiner function 809
dump-form function 1067
dump-forms-to-file function 1067
dylib 191
dynamic libraries 191, 459
dynamic library 191
memory clash 194
relocation 194

E
:ed debugger command 18
editor
customizing 185
editor source code 186
editor-color-code-coverage function 917
ef-spec 443
:ef-spec initarg 1751
:element-type initarg 613, 1682
Emacs 3
enable-sql-reader-syntax function 346, 370, 1567
encode-db-standard-date function 1548
encode-db-standard-timestamp function 1548
encode-lisp-string function 835
encoding
changing default for files 449
enlarge-generation function 144, 146, 920
enlarge-static function 921
ensure-hash-entry function 922
ensure-is-jobject function 1247
ensure-lisp-classes-from-tree function 1233
ensure-loads-after-loads function 272, 1746
ensure-memory-after-store function 1746
ensure-objc-initialized function 243
ensure-process-cleanup function 1389
ensure-remote-debugging-connection function 36, 741
ensure-sal function 431, 577
ensure-stores-after-loads function 1747
execute-actions macro 1129
exception handlers
in applications 756
exception handling
for action lists 96
exceptions
handling 756
executable 454
filename 454, 1149
pathname 454, 1149
execute-command function 352, 1567
execute-with-interface function 267
execution functions 95
execution profiling 167
execution stack
java.lang.object
*enter-debugger-directly*
variable 1124
:entrycond keyword 52
enum-registry-value function 468, 1929
environment access API 857, 901, 940, 1059
environment-variable function 1125
eql-specializer class 250
eql-specializer-object function 250
errno-value function 1126
:error debugger command 18
error handlers
in applications 756
error handling
in parser generator 316
error output 1810
*error-output* variable 705
errors in Common SQL 372
error-situation-forms macro 923
EUC-JP 444
:euc-jp external format 444
:eval-after keyword 51
:eval-before keyword 50
evaluating
forms during tracing 49–51
example-compile-file function 1127
example-edit-file function 1128
example-file function 1127
example-load-binary-file function 1129
exception handlers
in applications 756
exception handling
for action lists 96
exceptions
handling 756
executable 454
filename 454, 1149
pathname 454, 1149
execute-command function 352, 1567
execute-with-interface function 267
execution functions 95
execution profiling 167
execution stack
Index

examine 12
:exitcond keyword 52
expand-generation-1 function 146, 924
extend-current-stack function 925
extended-char type 1131
extended-character type 1131
extended-character-p function 1132
*extended-spaces* variable 1203, 1748
extended-time macro 149, 163, 173, 926
external format
changing default for files 449
external format specification 443
external formats 443
:ascii 444
:bmp 444, 827
:bmp-native 446, 827
:bmp-reversed 446, 827
code-page 443
:euc-jp 444
:gbk 445
:jis 444
:koi-8 445
:latin-1 443
:latin-1-safe 444
:latin-1-terminal 443
:macos-roman 444
:sjis 444
:unicode 444
:utf-16 444
:utf-16be 446, 830
:utf-16le 446, 830
:utf-16-native 446, 830
:utf-16-reversed 446, 830
:utf-32 444
:utf-8 444
:windows-cp936 444
external programs
calling from Lisp 462
external-format-error condition 835
external-format-foreign-type function 836
*external-formats* variable 1133
external-format-type function 836
:extra-initargs class option 487,

F
false function 1134
fasl (fast load)
description 101
fasl-error condition 929
fast-directory-files function 930
FDDL 353
fdf-handle-directory-p accessor 930
fdf-handle-directory-string accessor 930
fdf-handle-last-access accessor 930
fdf-handle-last-modify accessor 930
fdf-handle-link-p accessor 930
fdf-handle-size accessor 930
fdf-handle-writable-p accessor 930
FDDL 357
*features* variable 670
field-access-exception condition 1236
field-access-exception-set-p accessor 1236
field-accession-condition 1237
field-accession-class-name accessor 1237
field-accession-field-name accessor 1237
file descriptor, of socket-stream 613
file descriptor, on UNIX 1436, 1761
file-directory-p function 1134
*file-encoding-detection-algorithm* variable 448, 1749
file-encoding-resolution-error condition 1751
*file-eol-style-detection-algorithm* variable 448, 1751
filename of deliverable 454, 1149
filename of DLL 454, 1149
filename of dynamic library 1149
filename of executable 454, 1149
filename of lisp image 454, 1149
*filename-pattern-encoding-matches* variable 1752
files
load-on-demand 188
file-stat-blocks function 1762
file-stat-device function 1761
file-stat-device-type function 1762
file-stat-group-id function 1762
file-stat-inode function 1761
file-stat-last-access function 1762
file-stat-last-change function 1762
file-stat-last-modify function 1762
file-stat-links function 1762
file-stat-mode function 1762
file-stat-owner-id function 1761
file-stat-size function 1762
file-string function 932
file-writable-p function 933
(setq fill-pointer) function 266
filter-code-coverage-data function 933
Find Source
menu command 799
Find Source Editor command 186, 1222
Find Source For Dspec Editor command 816
find-database function 338, 1568
find-dspec-locations function 809
find-encoding-option function 1752
find-external-char function 837
find-filename-pattern-encoding-match function 1753
find-java-class function 1238
find-name-locations function 810
find-object-size function 145, 935
find-process-from-name function 1391
find-regexp-in-string function 1135
find-throw-tag function 936
finish-heavy-allocation function 936
fixnum type 477
fixnum-safety 105
flag-not-special-free-action function 165, 937
flag-special-free-action function 164, 938
FLI types
lpcstr 1916
lpctstr 1917
lpcwstr 1918
lpwstr 1918
lpstr 1916
lptstr 1917
p-oci-env 385, 1625
p-oci-file 385, 1626
p-oci-lob-locator 385, 1626
p-oci-lob-or-file 1626
p-oci-svc-ctx 385, 1626
ssl-cipher-pointer 427, 622
ssl-cipher-pointer-stack 622
ssl-ctx-pointer 427, 624
ssl-pointer 427, 625
str 1916
tstr 1917
wstr 1918
float 105
float calculations, optimizing 112
foreign callbacks 297
foreign-slot-value function 116
foreign-symbol-address function 1076
format-to-java-host function 1239
forms
  evaluating when tracing 49–51
frame, examining 12
:free-lob-locator-on-close initarg 1577
:func debugger command 19
funcallable-standard-class class 251
funcallable-standard-instance-access function 250
funcallable-standard-object class 252, 494
funcall-async function 1392
funcall-async-list function 1392
function dspecs 84
function, altering with advice 63
Functional DDL 353
Functional DML 346
functional interface in Common SQL 346
function-information function 939
function-lambda-list function 1137
functions
  accepting-handle-collection 518
Index

accepting-handle-local-port 520
accepting-handle-name 519
accepting-handle-socket 520
accepting-handle-user-info 521
accepts-n-syntax 1527
accept-tcp-connections-creating-async-io-states 412, 416, 422, 513
add-code-coverage-data 839
add-special-free-action 164, 841
add-sql-stream 372, 1528
add-symbol-profile 842
allocated-in-its-own-segment-p 1715
android-build-value 848
android-funcall-in-main-thread 849
android-funcall-in-main-thread-list 849
android-get-current-activity 851
android-main-thread-p 852
any-capi-window-displayed-p 853
any-other-process-non-internal-server-p 1361
append-file 1082
apply-in-wait-state-collection-process 411, 522
apply-with-allocation-in-gen-num 147, 1716
apropos 639
apropos-list 640
array-single-thread-p 854
array-weak-p 854
async-io-state-abort 413, 417, 523
async-io-state-abort-and-close 417, 525
async-io-state-address 526
async-io-state-attach-ssl 422, 425, 526
async-io-state-buffered-data-length 414, 528
async-io-state-collection 530
async-io-state-ctx 431, 529
async-io-state-detach-ssl 425, 529
async-io-state-discard 413, 417, 531
async-io-state-finish 413, 417, 532
async-io-state-get-buffered-data 414, 417, 533
async-io-state-handshake 534
async-io-state-max-read 535
async-io-state-name 536
async-io-state-object 536
async-io-state-old-length 537
async-io-state-peer-address 538
async-io-state-read-buffer 413, 416, 538
async-io-state-read-status 540
async-io-state-read-timeout 541
async-io-state-read-with-checking 413, 416, 542
async-io-state-receive-message 413, 416, 545
async-io-state-send-message 413, 416, 547
async-io-state-send-message-to-address 413, 416, 549
async-io-state-ssl 431, 551
async-io-state-ssl-side 431, 551
async-io-state-user-info 412, 553
async-io-state-write-buffer 413, 416, 552
async-io-state-write-status 540
async-io-state-write-timeout 541
attach-ssl 420, 422, 554
attribute-type 353, 1529
augmented-string-p 1722
augment-environment 855
avoid-ge 146, 857
barrier-arriver-count 1362
barrier-block-and-wait 1363
barrier-change-count 1366
barrier-count 1367
barrier-disable 287, 1368
barrier-enable 287, 1368
barrier-name 1369
barrier-pass-through 1370
barrier-unblock 1370
barrier-wait 287, 1372
base-character-p 1084
base-char-p 1085
base-char-ref 1821
base-string-p 1086
binds-who 860
bmp-char-p 1087
bmp-string-p 1089
break-new-instances-on-access 485
break-on-access 486
break-on-unresolved 1075
building-universal-intermediate-p 862
cache-table-queries 349, 1531
call-java-method 1209
call-next-advice 66, 73, 1090
calls-who 863
call-system 1724
call-system-showing-output 1727
call-wait-state-collection 411, 413, 556
canonicalize-dspec 793
canonicalize-sid-string 1895
cdr-assoc 1730
change-directory 187, 864
change-process-priority 295, 1375
char-external-code 833
checked-read-java-field 1302
check-fragmentation 145, 146, 865
check-java-field 1302
check-lisp-calls-initialized 1213
choose-unicode-string-hash-function 1092
clean-down 136, 163, 866
clean-generation-0 146, 868
clear-code-coverage 869
close-accepting-handle 557
close-async-io-state 414, 415, 417, 558
close-registry-key 468, 1923
close-remote-debugging-connection 36, 729
close-serial-port 1519
close-wait-state-collection 411, 559
code-coverage-data-generator-coloring-html 871
code-coverage-data-generation-statistics 877
code-coverage-set-editor-colors 881
code-coverage-set-editor-default-data 883
code-coverage-set-html-background-colors 884
coerce 642
coerce-to-gesture-spec 1731
collect-generation-2 144, 146, 885
collect-highest-generation 146, 886
collect-registry-subkeys 468, 1924
collect-registry-values 468, 1925
commit 348, 350, 382, 1532
compile 643
compile-file 645
compile-file-if-needed 887
compile-system 306, 1092
concatenate 650
concatenate-system 1094
condition-variable-broadcast 286, 1376
condition-variable-signal 286, 1377
condition-variable-wait 285, 1378
condition-variable-wait-count 286, 1379
configure-remote-debugging-spec 26, 27, 33, 730
connect 338, 339, 374, 1533
connected-databases 338, 1542
connect-to-named-pipe 1896
connect-to-tcp-server 572
copy-code-coverage-data 889
copy-current-code-coverage 889
copy-file 1096
copy-from-sqlite-raw-blob 394, 1655
copy-preferences-from-older-version 1734
copy-standard-object 493
copy-to-weak-simple-vector 165, 891
count-gen-num-
Index

allocation 150, 1735
count-regexp-occurrences 1097
create-and-run-wait-state-collection 411, 560
create-async-io-state 412, 415, 562
create-async-io-state-and-connected-tcp-socket 412, 422, 564
create-async-io-state-and-connected-udp-socket 413, 567
create-async-io-state-and-udp-socket 412, 569
create-client-remote-debugging-connection 28, 734
create-ide-remote-debugging-connection 734
create-index 354, 1542
create-instance-from-jobject 1213
create-instance-jobject 1214
create-instance-jobject-list 1214
create-java-object 1216
create-macos-application-bundle 892
create-registry-key 468, 1927
create-table 354, 1543
create-temp-file 467, 895
create-universal-binary 897
create-view 354, 1545
create-view-from-class 355, 1546
current-function-name 897
current-pathname 1099
current-process-block-interrupts 289, 1380
current-process-in-cleanup p 1381
current-process-kill 1382
current-process-pause 1383
current-process-send 1385
current-process-set-terminate-method 1386
current-process-unblock-interrupts 289, 1387
current-stack-length 898
database-name 338, 1546
date-string 899
dde-advice-start 1941
dde-advice-start* 1944
dde-advice-stop 1945
dde-advice-stop* 1947
dde-connect 1949
dde-disconnect 1950
dde-execute 1950
dde-execute* 1951
dde-execute-command 1952
dde-execute-command* 1953
dde-execute-string 1955
dde-execute-string* 1955
dde-poke 1960
dde-poke* 1962
dde-request 1964
dde-request* 1966
debug-other-process 1388
declaration-information 900
decode-external-string 834
decode-to-db-standard-date 1547
decode-to-db-standard-timestamp 1547
default-eol-style 1737
default-name-constructor 1218
delete-directory 1115
delete-duplicates 116
delete-records 348, 349, 1559
delete-registry-key 468, 1929
delete-sql-stream 372, 1560
deliver 191, 454, 460, 1116
delivered-image-p 908
deliver-to-android-project 909
describe 37, 663
destroy-prepared-statement 1561
destroy-ssl 431, 574
destroy-ssl-ctx 431, 575
destructive-add-code-coverage-data 839
destructive-reverse-subtract-code-coverage-data 839
destructive-subtract-code-coverage-data 839
detect-eol-style 1742
detect-ide-remote-debugging-command 743
detect-japanese-encoding-in-file 1743
detect-unicode-bom 1744
detect-utf32-bom 1744
Index

detect-utf8-bom 1744
directory 663
disable-sql-reader-
syntax 370, 1561
disassemble 668
discard-source-info 803
disconnect 338, 1562
dismiss-splash-screen 1898
dll-quit 195, 1120
do-nothing 1123
do-profiling 169, 915
do-rand-seed 576
dotted-list-length 1122
dotted-list-p 1123
drop-index 354, 1564
drop-table 354, 1565
drop-view 354, 1565
drop-view-from-class 355, 1566
dspec-class 803
dspec-defined-p 804
dspec-definition-
locations 805
dspec-equal 806
dspec-name 806
dspec-primary-name 807
dspec-progenitor 808
dspec-subclass-p 808
dspec-undefineder 809
dump-form 1067
dump-forms-to-file 1067
dump-forms-to-file 1067
dump-forms-to-file 1067
eeditor-color-code-
coverage 917
enable-sql-reader-
syntax 346, 370, 1567
encode-db-standard-date 1548
encode-db-standard-
timestamp 1548
encode-lisp-string 835
enlarge-generation 144, 146, 920
enlarge-static 921
ensure-hash-entry 922
ensure-is-jobject 1247
ensure-lisp-classes-from-
tree 1233
ensure-loads-after-
loads 272, 1746
ensure-memory-after-
store 1746
ensure-objc-initialized 243
ensure-process-cleanup 1389
ensure-remote-debugging-
connection 36, 741
ensure-ssl 431, 577
ensure-stores-after-
memory 1747
ensure-stores-after-
stores 269, 273, 275, 1748
ensure-supers-contain-
java.lang.object 1235
dispatch 468, 1929
directory 1125
dirname 250
dirname-value 1126
date-compile-file 1127
date-edit-file 1128
date-file 1127
date-load-binary-
file 1129
date-executable-log-
file 741
date-execute-command 352, 1567
date-expand-generation-1 146, 924
date-extend-current-stack 925
date-extended-character-p 1132
date-extended-character-p 1132
date-external-format-
foreign-type 836
date-external-format-type 836
date-false 1134
date-fast-directory-files 930
date-file-directory-p 1134
date-file-stat-blocks 1762
date-file-stat-device 1761
date-file-stat-device-type 1762
date-file-stat-group-id 1762
date-file-stat-inode 1761
date-file-stat-last-access 1762
date-file-stat-last-change 1762
date-file-stat-last-modify 1762
date-file-stat-links 1762
date-file-stat-mode 1762
date-file-stat-owner-id 1761
date-file-stat-size 1762
date-file-string 932
date-file-writable-p 933
date-filter-code-coverage-
data 933
date-find-database 338, 1568
date-find-dspec-locations 809
date-find-encoding-option 1752
date-find-external-char 837
date-find-filename-pattern-
encoding-match 1753
date-find-java-class 1238
date-find-name-locations 810
Index

find-object-size 145, 935
find-process-from-name 1391
find-regexp-in-string 1135
find-throw-tag 936
finish-heavy-allocation 936
flag-not-special-free-action 165, 937
flag-special-free-action 164, 938
foreign-slot-value 116
foreign-symbol-address 1076
format-to-java-host 1239
funcallable-standard-instance-access 250
funcall-async 1392
funcall-async-list 1392
function-information 939
function-lambda-list 1137
gc-generation 136, 138, 146, 150, 942
gc-if-needed 146, 946
generate-code-coverage 946
generate-java-class-definitions 1239
generation-number 144, 1755
gen-num-segments-fragmentation-state 150, 1754
genSYM 164
gesture-spec-data 1758
gesture-spec-modifiers 1758
gesture-spec-p 1759
gesture-spec-to-character 1760
get-code-coverage-delta 940
get-current-process 262, 263, 1394
get-default-generation 146, 949
get-default-local-ipv6-address 578
get-file-stat 1761
get-folder-path 467, 1762
get-foreign-symbol 1077
get-form-parser 811
get-gc-parameters 146, 950
get-gc-timing 1031
gethash-ensuring 952
get-host-entry 578
get-host-java-virtual-machine 1245
get-ip-default-zone-id 581
get-java-virtual-machine 1246
get-object 1247
get-maximum-allocated-in-generation-2-after-gc 1765
get-primitive-array-region 1248
get-process 263, 1395
get-process-private-property 1396
get-serial-port-state 1520
get-service-entry 582
go-socket-address 583
go-socket-peer-address 584
go-superclass-and-interfaces-tree 1250
go-temp-directory 951
get-unix-error 1142
go-user-profile-directory 467, 1767
get-verification-mode 585
go-working-directory 952
guess-external-format 448, 1769
hardcopy-system 1147
go-ide-attach-remote-output-stream 30, 744
ide-connect-remote-debugging 27, 28, 745
ide-eval-form-in-remote 29, 30, 747
ide-find-remote-debugging-connection 35, 750
ide-funcall-in-remote 29, 30, 747
ide-list-remote-debugging-connections 35, 750
ide-open-a-listener 25, 28, 753
ide-set-default-remote-debugging-connection 35, 750
ide-set-remote-symbol-value 30, 747
immediatep 1770
initialize-database-type 1569
initialize-database-type 263, 1397
init-java-interface 1253
insert-records 348, 381, 1570
inspect 38
int32* 1773
Index

int32+ 473, 1773
int32- 473, 1773
int32/ 1773
int32/= 1775
int32< 1775
int32<< 1776
int32<= 1775
int32= 1775
int32>> 1776
int32-1+ 1774
int32-1- 1774
int32-incorrect 1777
int32-logand 1778
int32-logandc1 1778
int32-logandc2 1778
int32-logbitp 1778
int32-logeqv 1778
int32-logior 1778
int32-logiorc1 1778
int32-logiorc2 1778
int32-lognot 1778
int32-lognotc1 1778
int32-lognotc2 1778
int32-logor 1778
int32-logorb1 1778
int32-logorb2 1778
int32-logtest 1778
int32-logxor 1778
int32-logxor 1778
int32-minusp 1781
int32-plusp 1781
int32-to-int64 1781
int32-to-integer 1782
int32-zerop 1781
int64- 474, 1784
int64- 474, 1784
int64/= 1784
int64< 1786
int64<< 1787
int64<= 1786
int64= 1786
int64>= 1786
int64=> 1786
int64-1+ 1785
int64-1- 1785
int64-incorrect 1788
int64-logand 1789
int64-logandc1 1789
int64-logandc2 1789
int64-logbitp 1789
int64-logeqv 1789
int64-logior 1789
int64-lognot 1789
int64-logor 1789
int64-logror 1789
int64-logtest 1789
int64-logxor 1789
int64-minusp 1791
int64-plusp 1791
int64-to-int32 1792
int64-to-integer 1793
int64-zerop 1791
integer-to-int32 1793
integer-to-int64 1794
interactive-stream-p 676
intern-and-export-list 1260
intern-eql-specializer 250
ip-address-string 585
ipv6-address 586
ipv6-address-p 587
ipv6-address-scope-id 587
ipv6-address-string 588
jaref 1260
java-array-element-type 1262
java-array-length 1264
java-field-class-name-for-setting 1302
java-object-array-element-type 1275
java-objects-eq 1275
java-primitive-array-element-type 1277
java-type-to-lisp-array-type 1279
object-class-name 1282
object-ensure-global 1283
object-of-class-p 1284
object-p 1285
object-pretty-class-name 1285
object-string 1286
object-to-lisp 1287
jvref 1288
known-sid-integer-to-sid-string 1901
last-callback-on-thread 1399
lisp-array-to-primitive-array 1299
lisp-array-type-to-java-type 1278
lisp-image-name 454, 1149

2013
Index

lisp-java-instance-p 1290
lisp-name-to-foreign-name 1078
lisp-to-jobject 1290
list-all-processes 263, 1400
list-attributes 353, 1574
list-attribute-types 353, 1573
list-classes 357, 1575
list-sql-streams 372, 1576
list-tables 353, 1576
load-all-patches 1152
load-code-coverage-data 889
load-data-file 957
load-logical-pathname-translations 677
load-system 1153
lob-stream-lob-locator 384
locale-dspec-p 812
locally-disable-sql-reader-syntax 370, 1578
locally-enable-sql-reader-syntax 370, 1579
lock-and-condition-variable-broadcast 1401
lock-and-condition-variable-signal 1402
lock-and-condition-variable-wait 1404
lock-locked-p 1406
lock-name 277, 1409
lock-owned-by-current-process-p 1407
lock-owner 277, 1409
lock-recursively-locked-p 1408
lock-recursive-p 1407
log-bug-form 754
logs-directory 755
long-namestring 454, 1903, 1915
long-site-name 453, 678
loop-processing-wait-state-collection 411, 413, 589
low-level-atomic-place-p 1797
mailbox-count 1411
mailbox-empty-p 1412
mailbox-full-p 1413
mailbox-not-empty-p 1414
mailbox-peek 1414
mailbox-read 1415
mailbox-reader-process 1416
mailbox-send 1417
mailbox-send-limited 1418
mailbox-size 1420
mailbox-wait 1421
mailbox-wait-for-event 1422
make-array 165, 276, 680
make-barrier 287, 1424
make-condition-variable 1426
make-current-allocation-permanent 1798
make-gesture-spec 1800
make-hash-table 165, 276, 682
make-java-array 1291
make-java-instance 1292
make-lisp-proxy 1293
make-lisp-proxy-with-overrides 1293
make-lock 1427
make-mailbox 1429
make-mt-random-state 1156
make-named-timer 1430
make-object-permanent 1805
make-permanent-simple-vector 1807
make-ring 962
make-semaphore 288, 1431
make-sequence 685
make-simple-int32-vector 1808
make-simple-int64-vector 1809
make-ssl-ctx 422, 423, 588
make-stderr-stream 1810
make-string 686
make-string-output-stream 687
make-symbol 164
make-timer 293, 1432
make-typed-aref-vector 1811
make-unlocked-queue 964
make-unregistered-action-list 1154
make-wait-state-collection 411, 590
map 688
map-all-processes 1433
map-all-processes-backtrace 1434
map-code-coverage-data 966
map-environment 1812
map-java-object-array 1295
map-process-backtrace 1434
map-processes 1435
map-query 351, 381, 1582
map-ring 967
mark-and-sweep 967
marking-gc 150, 1813
memory-growth-margin 146, 1815
merge 688
merge-code-coverage-data 970
merge-ef-specs 1816
mobile-gc-p 1817
mobile-gc-sweep-objects 1818
modify-hash 971
mt-random 1157
mt-random-state-p 1158
name-defined-dspecs 813
name-definition-locations 814
named-pipe-stream-name 1904
name-only-form-parser 815
normal-gc 146, 972
notice-fd 1436
object-address 1819
object-dspec 816
object-pointer 1820
octet-ref 1821
open 689
open-named-pipe-stream 1905
open-pipe 1823
open-registry-key 468, 1932
open-serial-port 1517
openssl-version 599
open-tcp-stream 420, 422, 591
open-tcp-stream-using-java 595
open-temp-file 467, 895
open-url 1827
ora-lob-append 388, 1586
ora-lob-assign 387, 1587
ora-lob-char-set-form 386, 1587
ora-lob-char-set-id 1588
ora-lob-close 388, 1589
ora-lob-copy 388, 1590
ora-lob-create-empty 381, 387, 1591
ora-lob-create-temporary 389, 1592
ora-lob-disable-buffering 389, 1593
ora-lob-element-type 386, 1594
ora-lob-enable-buffering 389, 1595
ora-lob-env-handle 384, 1596
ora-lob-erase 388, 1596
ora-lob-file-close 388, 1597
ora-lob-file-close-all 388, 1598
ora-lob-file-exists 1599
ora-lob-file-get-name 1600
ora-lob-file-is-open 1600
ora-lob-file-open 388, 1601
ora-lob-file-set-name 388, 1602
ora-lob-flush-buffer 389, 1603
ora-lob-free 387, 1604
ora-lob-free-temporary 389, 1605
ora-lob-get-buffer 384, 389, 1605
ora-lob-get-chunk-size 387, 1608
ora-lob-get-length 387, 1608
ora-lob-internal-lob-p 386, 387, 1609
ora-lob-is-equal 387, 1610
ora-lob-is-open 387, 1610
ora-lob-is-temporary 387, 389, 1611
ora-lob-load-from-file 388, 1612
ora-lob-lob-locator 384, 1613
ora-lob-locator-is-init 387, 1614
ora-lob-open 388, 1614
ora-lob-read-buffer 386, 389, 1615
ora-lob-read-foreign-buffer 384, 386, 389, 1619
ora-lob-read-into-plain-file 389, 1618
ora-lob-svc-ctx-handle 384, 1620
ora-lob-trim 388, 1620
ora-lob-write-buffer 386, 389, 1621, 1624
ora-lob-write-foreign-buffer 384, 386, 389
ora-lob-write-from-plain-file 389, 1623
output-backtrace 756
package-flagged-p 1827
parse-float 975
parse-form-dspec 817
pathname-location 1159
Index

pem-read 426, 602
pipe-close-connection 1828
pipe-exit-status 1829
pipe-kill-process 1830
pointer-from-address 1832
position-in-ring 976
position-in-ring-forward 976
precompile-regexp 1160
prepare-statement 1627
primitive-array-to-lisp-array 1299
print-action-lists 97, 1161
print-actions 97, 1161
print-pretty-gesture-spec 1833
print-profile-list 171, 977
process-alive-p 1436
process-allow-scheduling 262, 1438
process-arrest-reasons 1438
process-break 1439
process-continue 1439
process-count 263, 1482
process-exclusive-lock 1440
process-exclusive-unlock 1441
process-idle-time 1442
process-internal-server-p 1444
process-interrupt 290, 1445
process-interrupt-list 1446
process-join 1447
process-kill 1448
process-lock 277, 1449
process-mailbox 1450
process-name 263, 1450
process-p 1451
process-plist 294, 1451
process-poke 1452
process-priority 295, 1454
process-private-property 1455
process-property 294, 1456
process-reset 1457
process-run-function 262, 1458
process-run-reasons 1461
process-run-time 1462
process-send 1463
process-sharing-lock 1465
process-sharing-unlock 1466
process-stop 296, 1467
process-stopped 296
process-stopped-p 1468
process-terminate 1468
process-unlock 1470
process-unstop 296, 1471
process-wait 285, 1472
process-wait-for-event 1473
process-wait-function 1474
process-wait-local 1474
process-wait-local-with-periodic-checks 1477
process-wait-local-with-timeout 1478
process-wait-local-with-timeout-and-periodic-checks 1479
process-who-state 1482
proclaim 104, 109, 693
product-registry-path 464, 1835
profiler-tree-from-function 983
profiler-tree-to-function 984
ps 263, 1485
pushnew-to-process-private-property 294, 1483
pushnew-to-process-property 294, 1484
query 352, 380, 1631
query-registry-key-info 468, 1933
query-registry-value 468, 1934
quit 4, 463, 1165
read-dhparams 426, 603
read-foreign-modules 1078
read-java-field 1302
read-serial-port-char 1521
read-serial-port-string 1521
reconnect 338, 1632
record-definition 817
record-java-class-lisp-symbol 1304
record-message-in-windows-event-log 1909
reduce-memory 164, 985
references-who 987
regexp-find-symbols 1168
registry-key-exists-p 468,
1935
registry-value 468, 1935
remote-debugging-connection-add-close-cleanup 36, 765
remote-debugging-connection-name 36, 767
remote-debugging-connection-peer-address 36, 768
remote-debugging-connection-remove-close-cleanup 36, 765
remote-inspect 25, 27, 28, 34, 770
remote-object-connection 29, 35, 772
remote-object-p 29, 772
remote-advice 66, 73, 1169
remote-duplicates 116
remote-from-process-private-property 294, 1486
remote-from-process-property 294, 1486
remote-process-private-property 294, 1487
remote-process-property 294, 1488
remove-special-free-action 165, 988
remove-symbol-profile 989
replace-from-sqlite-blob 395, 1650
replace-from-sqlite-raw-blob 395, 1655
replace-into-sqlite-blob 395, 1650
replace-socket-stream-socket 604
replace-standard-object 499
report-error-to-java-host 1305
reset-code-coverage 869
reset-code-coverage-snapshot 940
reset-java-interface-for-new-jvm 1306
reset-profile 989
reset-ring 990
restore-code-coverage-data 869
restore-sql-reader-syntax-state 370, 1633
results for traced 50
reverse-subtract-code-coverage-data 839
ring-length 991
ring-name 992
ringp 991
ring-pop 992
ring-push 993
ring-ref 994
rollback 348, 350, 382, 1634
room 145, 146, 149, 458, 695
room-values 458, 1838
rotate-byte 1172
rotate-ring 995
round-to-single-precision 1172
run-shell-command 1839
safe-format-to-limited-string 996
safe-format-to-string 996
safe-locale-file-encoding 1844
safe-print-to-string 996
safe-prin-to-string 996
save-argument-real 997
save-code-coverage-data 889
save-current-code-coverage 889
save-current-profile-tree 174, 998
save-current-session 1000
save-image 179, 191, 454, 460, 464, 1001
save-image-with-bundle 1010
save-tags-database 821
save-universal-from-script 1011
sbchar 1173
schedule-timer 293, 1489
schedule-timer-milliseconds 1491
schedule-timer-relative 1493
schedule-timer-relative-milliseconds 1494
security-description-string-for-open-named-pipe 1910
select 346, 357, 1634
semaphore-acquire 288, 1496
semaphore-count 288, 1498
semaphore-name 288, 1498
semaphore-release 288, 1499
semaphore-wait-count 288, 1500
Index

send-message-to-java-host 1307
sequencep 1174
serial-port 1520
serial-port-input-available-p 1522
server-terminate 605
set-application-themed 1914
set-approaching-memory-limit-callback 1845
set-array-single-threaded 1012
set-array-weak 165, 1012
set-automatic-gc-callback 150, 1846
set-blocking-gen-num 150, 1848
set-clos-initarg-checking 501
set-code-coverage-snapshot 940
set-debugger-options 773
set-default-character-element-type 439, 442, 443, 1174
set-default-generation 137, 146, 1014
set-default-remote-debugging-connection 44, 774
set-default-segment-size 151, 1851
set-delay-promotion 151, 1852
set-expected-allocation-in-generation-2-after-gc 1853
setf cdr-assoc 1730
setf timer-name 1503
set-file-dates 1857
set-funcall-async-limit 1500
set-gc-parameters 139, 146, 1015
set-generation-2-gc-options 1858
set-gen-num-gc-threshold 150, 1860
set-hash-table-weak 165, 1017
set-java-field 1302
set-make-instance-argument-checking 502
set-maximum-memory 146, 1862
set-maximum-segment-size 148, 150, 1863
set-memory-check 1865
set-memory-exhausted-callback 1866
set-minimum-free-space 139, 146, 1019
set-prepared-statement-variables 1639
set-primitive-array-region 1248
set-process-profiling 169, 170, 1020
set-promoter-threshold 1022
set-promote-generation-1 1868
set-promotion-count 1023
set-quit-when-no-windows 1176
set-registry-value 468, 1937
set-remote-debugging-connection 33, 775
set-reserved-memory-policy 1869
set-serial-port-state 1523
set-signal-handler 1870
set-spare-keeping-policy 151, 1871
set-split-promotion 1872
set-ssl-ctx-dh 426, 608
set-ssl-ctx-options 426, 610
set-ssl-ctx-password-callback 426, 610
set-ssl-library-path 420, 611
set-static-segment-size 1874
sets-who 1030
set-system-message-log 1024
set-temp-directory 468, 1875
setup-atomic-funcall 1876
setup-deliver-dynamic-library-for-java 217, 1308
setup-field-accessor 1312
setup-java-caller 1313
setup-java-constructor 1313
setup-java-interface-callbacks 1253
setup-lisp-proxy 1315
set-up-profiler 168, 1025
set-verification-mode 606
short-namestring 454
short-site-name 453, 701
sid-string-to-user-name 1916
simple-augmented-string-p 1722
simple-base-string-p 1086
simple-bmp-string-p 1089
simple-char-p 1177
simple-int32-vector-length 1879
simple-int32-vector-p 1879
simple-int64-vector-length 1880
simple-int64-vector-p 1881
simple-lock-and-condition-variable-wait 1501
simple-text-string-p 1186
single-form-form-parser 822
single-form-with-options-form-parser 822
socket-stream-address 617
socket-stream-ctx 431, 618
socket-stream-handshake 618
socket-stream-peer-address 619
socket-stream-shutdown 620
socket-stream-ssl 431, 621
socket-stream-ssl-side 431, 622
software-type 453, 703
software-version 453, 704
source-debugging-on-p 1031
specific-valid-file-encoding 1882
split-sequence 1177
split-sequence-if 1178
split-sequence-if-not 1179
sql 368, 1642
sql-expression 368, 1645
sqlite-blob-length 1650
sqlite-blob-p 1650
sqlite-close-blob 395, 1650
sqlite-last-insert-rowid 1649
sqlite-open-blob 395, 1650
sqlite-raw-blob-length 395, 1655
sqlite-raw-blob-p 1655
sqlite-raw-blob-ref 395, 1655
sqlite-raw-blob-valid-p 395, 1655
sqlite-reopen-blob 1650
sql-operation 368, 1660
sql-operator 368, 1662
sql-recording-p 372, 1663
sql-stream 372, 1664
ssl-add-client-ca 427
ssl-cipher-get-bits 427
ssl-cipher-get-name 427
ssl-cipher-get-version 427
ssl-close-num-renegotiations 427
ssl-ctrl 428
ssl-ctx-add-client-ca 428
ssl-ctx-add-extra-chain-cert 428
ssl-ctx-ctrl 428
ssl-ctx-get-max-cert-list 428
ssl-ctx-get-mode 428
ssl-ctx-get-options 428
ssl-ctx-get-read-ahead 428
ssl-ctx-get-verify-mode 428
ssl-ctx-load-verify-locations 428
ssl-ctx-need-temp-rsa 428
ssl-ctx-sess-get-cache-mode 428
ssl-ctx-sess-get-cache-size 428
ssl-ctx-sess-set-cache-mode 428
ssl-ctx-sess-set-cache-size 428
ssl-ctx-set-client-ca-list 428
ssl-ctx-set-max-cert-list 428
ssl-ctx-set-mode 428
ssl-ctx-set-options 428
ssl-ctx-set-read-ahead 428
ssl-ctx-set-temp-dh 428
ssl-ctx-set-temp-rsa 428
ssl-ctx-use-certificate-chain-file 428
ssl-ctx-use-certificate-file 429
ssl-ctx-use-privatekey-file 429
ssl-ctx-use-rsa-privatekey-file 429
ssl-get-current-cipher 429
ssl-get-max-cert-list 429
ssl-get-mode 429
ssl-get-options 429
ssl-get-verify-mode 429
ssl-get-version 429
ssl-load-client-ca-file 429
ssl-need-temp-rsa 429
ssl-new 431, 625
ssl-num-renegotiations 429
ssl-session-reused 429
ssl-set-accept-state 424, 429
ssl-set-client-ca-list 429
ssl-set-connect-state 424, 429
ssl-set-max-cert-list 429
ssl-set-mode 429
ssl-set-options 429
ssl-set-tmp-dh 429
ssl-set-tmp-rsa 429
ssl-total-renegotiations 429
ssl-use-certificate-file 430
ssl-use-privatekey-file 430
ssl-use-rsaprivatekey-file 430
standard-instance-access 250
start-client-remote-debugging-server 27, 33, 776
start-dde-server 1983
start-gc-timing 1031
start-ide-remote-debugging-server 26, 27, 779
start-profiling 169, 170, 1033
start-remote-listener 25, 27, 28, 34, 781
start-sql-recording 372, 1666
start-tty-listener 1180
start-up-server 626
start-up-server-and-mp 632
staticp 1184
status 338, 1667
stchar 1181
stop-gc-timing 1031
stop-profiling 169, 170, 1036
stop-sql-recording 372, 1668
string-append 1182
string-append* 1183
string-ip-address 633
string-needs-n-prefix 1668
string-prefix-with-n-if-needed 1669
string-trim-whitespace 1035
structurep 1184
subtract-code-coverage-data 839
sweep-all-objects 164, 1037
sweep-gen-num-objects 1885
switch-open-tcp-stream-with-ssl-to-java 634
switch-static-allocation 136, 137, 1038
symbol-dynamically-bound-p 1039
symeval-in-process 295, 1503
table-exists-p 1670
text-string-p 1186
throw-an-exception 1317
timer-expired-p 1504
timer-name 1505
toggle-source-debugging 117, 1040
total-allocation 145, 1042
traceable-dspec-p 823
trace-new-instances-on-access 505
trace-on-access 506
tracing-enabled-p 824
tracing-state 825
true 1186
truename 720
try-compact-in-generation 144, 146, 1051
try-move-in-generation 144, 146, 1052
typed-aref 1886
unbreak-new-instances-on-access 509
unbreak-on-access 509
undefine-declaration 1054
unicode-alpha-char-p 451, 1188
unicode-alphanumericp 451, 1189
unicode-both-case-p 451, 1190
unicode-char-equal 451, 1192
unicode-char-greaterp 451, 1192
unicode-char-lessp 451, 1192
unicode-char-not-equal 451, 1191
unicode-char-not-greaterp 451, 1193
unicode-char-not-lessp 451, 1193
unicode-lower-case-p 451, 1194
unicode-string-equal 451, 1194
unicode-string-greaterp 451, 1196
unicode-string-lessp 451, 1196
unicode-string-not-equal 451, 1191
unicode-string-not-greaterp 451, 1197
unicode-string-not-lessp 451, 1197
unicode-upper-case-p 451, 1198
unlocked-queue-count 964
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>unlocked-queue-peek</td>
<td>964</td>
</tr>
<tr>
<td>unlocked-queue-read</td>
<td>964</td>
</tr>
<tr>
<td>unlocked-queue-ready</td>
<td>964</td>
</tr>
<tr>
<td>unlocked-queue-size</td>
<td>964</td>
</tr>
<tr>
<td>unnotice-fd</td>
<td>1506</td>
</tr>
<tr>
<td>unschedule-timer</td>
<td>1506</td>
</tr>
<tr>
<td>untrace-new-instances-on-access</td>
<td>510</td>
</tr>
<tr>
<td>untrace-on-access</td>
<td>510</td>
</tr>
<tr>
<td>update-instance-for-different-class</td>
<td>722</td>
</tr>
<tr>
<td>update-instance-for-redefined-class</td>
<td>723</td>
</tr>
<tr>
<td>update-objects-joins</td>
<td>1672</td>
</tr>
<tr>
<td>update-records</td>
<td>348, 349, 381, 1673</td>
</tr>
<tr>
<td>user-homedir-pathname</td>
<td>227, 466, 1764</td>
</tr>
<tr>
<td>user-name-to-sid-string</td>
<td>1919</td>
</tr>
<tr>
<td>user-preference</td>
<td>464, 1199</td>
</tr>
<tr>
<td>valid-external-format-p</td>
<td>838</td>
</tr>
<tr>
<td>variable-information</td>
<td>1058</td>
</tr>
<tr>
<td>vector-pop</td>
<td>276</td>
</tr>
<tr>
<td>vector-push</td>
<td>276</td>
</tr>
<tr>
<td>vector-push-extend</td>
<td>276</td>
</tr>
<tr>
<td>verify-java-caller</td>
<td>1320</td>
</tr>
<tr>
<td>verify-java-callers</td>
<td>1321</td>
</tr>
<tr>
<td>verify-lisp-proxies</td>
<td>1324</td>
</tr>
<tr>
<td>verify-lisp-proxy</td>
<td>1324</td>
</tr>
<tr>
<td>wait-for-input-streams</td>
<td>1888</td>
</tr>
<tr>
<td>wait-for-input-streams-returning-first</td>
<td>1889</td>
</tr>
<tr>
<td>wait-for-state-collection</td>
<td>411, 635</td>
</tr>
<tr>
<td>wait-processing-events</td>
<td>1508</td>
</tr>
<tr>
<td>wait-serial-port-state</td>
<td>1524</td>
</tr>
<tr>
<td>wait-state-collection-stop-loop</td>
<td>411, 415, 636</td>
</tr>
<tr>
<td>whitespace-char-p</td>
<td>1203</td>
</tr>
<tr>
<td>who-binds</td>
<td>1059</td>
</tr>
<tr>
<td>who-calls</td>
<td>1060</td>
</tr>
<tr>
<td>who-references</td>
<td>1061</td>
</tr>
<tr>
<td>who-sets</td>
<td>1061</td>
</tr>
<tr>
<td>with-remote-debugging-connection</td>
<td>33</td>
</tr>
<tr>
<td>write-java-class-definitions-to-file</td>
<td>1326</td>
</tr>
<tr>
<td>write-java-class-definitions-to-stream</td>
<td>1326</td>
</tr>
<tr>
<td>write-serial-port-char</td>
<td>1524</td>
</tr>
<tr>
<td>write-serial-port-string</td>
<td>1525</td>
</tr>
<tr>
<td>write-string-with-properties</td>
<td>1072</td>
</tr>
<tr>
<td>yield</td>
<td>1514</td>
</tr>
<tr>
<td>fundamental-binary-input-stream-class</td>
<td>1683</td>
</tr>
<tr>
<td>fundamental-binary-output-stream-class</td>
<td>1684</td>
</tr>
<tr>
<td>fundamental-binary-stream</td>
<td>1684</td>
</tr>
<tr>
<td>fundamental-character-input-stream-class</td>
<td>400, 1685</td>
</tr>
<tr>
<td>fundamental-character-output-stream-class</td>
<td>400, 1686</td>
</tr>
<tr>
<td>fundamental-input-stream</td>
<td>1687</td>
</tr>
<tr>
<td>fundamental-output-stream</td>
<td>1688</td>
</tr>
<tr>
<td>fundamental-stream</td>
<td>1688</td>
</tr>
</tbody>
</table>

**Index**

G

- garbage collection, see also memory management
  - main chapter 129
  - GBK 445
  - :g bk external format 445
  - gc-generation function 136, 138, 146, 150, 942
  - gc-if-needed function 146, 946
  - general-handle-event generic function 1393
  - generate-code-coverage function 946
  - generate-java-class-definitions function 1239
  - generation definition 130
  - generation 2 144
  - generation-number function 144, 1755

generic functions

- accessor-method-slot-definition 250
- add-method 251
- class-extra-initargs 487
- close 641
- compute-applicable-methods-using-classes 251
- compute-class-potential-initargs 488
- compute-discriminating-function 251, 490
- compute-effective-method-
Index

function-from-classes 490
dde-client-advise-data 1948
dde-server-poke 1971
dde-server-request 1972
dde-server-topic 1973
dde-server-topics 1974
dde-topic-items 1976
default-constructor-arguments 1217
delete-instance-records 357, 1558, 1572
describe-object 38
documentation 668
genral-handle-event 1393
get-inspector-values 1138
input-stream-p 676
instance-refreshed 357
make-instance 684
make-method-lambda 250
open-stream-p 691
output-stream-p 692
print-object 189
process-a-class-option 495
process-a-slot-option 497
remote-debugging-stream-peer-address 769
slot-boundp-using-class 249, 502
slot-makunbound-using-class 249, 503
slot-value-using-class 249, 504
sort-inspector-p 1138
stream-advance-to-column 1689
stream-check-eof-no-hang 1690
stream-clear-input 402, 1690
stream-clear-output 403, 1691
stream-element-type 400, 709
stream-file-position 1692
stream-fill-buffer 1692
stream-finish-output 403, 1693
stream-flush-buffer 1694
stream-force-output 403, 1695
stream-fresh-line 1695
stream-line-column 403, 1696
stream-listen 402, 1697
stream-output-width 1698
stream-peek-char 1698
stream-read-buffer 1699
stream-read-byte 1700
stream-read-char 401, 1701
stream-read-char-no-hang 1701
stream-read-line 1702
stream-read-sequence 1703
stream-read-timeout 1704
stream-start-line-p 403, 1704
stream-terpri 1705
stream-unread-char 401, 1706
stream-write-buffer 1706
stream-write-byte 1707
stream-write-char 402, 1708
stream-write-sequence 1708
stream-write-string 1709
update-instance-from-records 357, 1671
update-record-from-instance 1674
update-record-from-slot 357, 1675
update-records-from-instance 357
wait-for-connection 1919
:gen-num initarg 1884
gen-num-segments-fragmentation-state 150, 1754
gensym function 164
gesture-spec-accelerator-bit variable 1756
gesture-spec-caps-lock-bit variable 1756
gesture-spec-control-bit variable 1756
gesture-spec-data function 1758
gesture-spec-hyper-bit variable 1756
gesture-spec-meta-bit variable 1756
gesture-spec-modifiers function 1758
gesture-spec-p function 1759
gesture-spec-shift-bit variable 1756
gesture-spec-super-bit variable 1756
gesture-spec-to-character function 1760
get 851
:get listener command 6
getApplicationContext Java
method 1355
getClassLoader Java method 1355
get-code-coverage-delta function 940
get-current-process function 262, 263, 1394
get-default-generation function 146, 949
get-default-local-ipv6-address function 578
get-file-stat function 1761
get-folder-path function 467, 1762
get-foreign-symbol function 1077
get-form-parser function 811
get-gc-parameters function 146, 950
get-gc-timing function 1031
gethash-ensuring function 952
get-host-entry function 578
get-host-java-virtual-machine function 1245
get-inspector-values generic function 1138
get-ip-default-zone-id function 581
get-java-virtual-machine function 1246
get-jobobject function 1247
get-maximum-allocated-in-generation-2-after-gc function 1765
get-primitive-array-region function 1248
GetProcAddress C function 191
get-process function 263, 1395
get-process-private-property function 1396
get-serial-port-state function 1520
get-service-entry function 582
get-socket-address function 583
get-socket-peer-address function 584
get-superclass-and-interfacetree function 1250
get-temp-directory function 951
get-unix-error function 1142
get-user-profile-directory function 467, 1767
get-verification-mode function 585
get-working-directory function 952
globally-accessible macro 269, 270, 273, 274, 1768
grammar
non-terminal 314
resolving ambiguities 315
rules 313
graphics ports liii
> SQL operator 363
*grep-command* variable 1142
*grep-command-format* variable 1143
*grep-fixed-args* variable 1144
guess-external-format function 448, 1769
GUI application 1002
H
:hi inspector command 39
*handle-existing-action-in-action-list* variable 1144, 96
*handle-existing-action-list* variable 1145, 96
*handle-existing-defpackage* variable 954
*handle-missing-action-in-action-list* variable 1146, 97
*handle-missing-action-list* variable 1145, 97
*handle-old-in-package* variable 955
*handle-old-in-package-used-as-make-package* variable 956
handler frame, examining 12
:handler keyword 761
*handle-warn-on-redefinition* variable 187, 1146
hardcopy-system function 1147
hash tables
weak 682
heap size
in 32-bit LispWorks 460, 478
in 64-bit LispWorks 461, 478
:help listener command 7
HFS+ filesystem 469
:hidden keyword 743
*hidden-packages* variable 24, 742
:his listener command 7
hook functions 95
Index

host 578
host name 578
hostname 578

I
:i inspector command 39
i8n 435
ide-attach-remote-output-stream function 30, 744
ide-connect-remote-debugging function 27, 28, 745
ide-eval-form-in-remote function 29, 30, 747
ide-find-remote-debugging-connection function 35, 750
ide-funcall-in-remote function 29, 30, 747
ide-list-remote-debugging-connections function 35, 750
ide-open-a-listener function 25, 28, 753
ide-remote-debugging system class 765
ide-set-default-remote-debugging-connection function 35, 750
ide-set-remote-symbol-value function 30, 747
image saving 179
image size 457
immediatep function 1770
impersonating-named-pipe-client macro 1899
impersonating-user macro 1900
import-java-class-definitions macro 1251
in cf macro 267
init file 1148
init Java method 1341
init_result_code Java method 1344
*init-file-name* variable 1148
initialization of Common SQL 334
initialization file 1148
initialize-database-type function 1569
*initialized-database-types* variable 1570
initialize-multiprocessing function 263, 1397
*initial-processes* variable 195, 263, 1398
init-java-interface function 1253
InitLispWorks C function 194, 460, 1985
input-stream-p generic function 676
insert-records function 348, 381, 1570
:inside keyword 55
inspect function 38
inspector main chapter 37
REPL 37
teletype 37
inspector commands :cv 39
:d 39
:dm 39
:dr 39
:h 39
:i 39
:m 39, 40
:q 39
:s 39
:sh 39
:u 39
:ud 39, 40
*inspect-print-length* variable 39
*inspect-print-level* variable 39
*inspect-through-gui* variable 1149
instance-refreshed generic function 357
in-static-area macro 137, 1771
int32* function 1773
int32+ function 473, 1773
int32- function 473, 1773
int32 type 472, 1772
int32/ function 1773
int32/= function 1775
int32<< function 1776
int32<< function 1775
int32= function 1775
int32>= function 1776
int32>= function 1775
int32> function 1775
int32>> function 1776
+int32-0+ symbol macro 1774
int32-1+ function 1774
int32-1- function 1774
<table>
<thead>
<tr>
<th>Symbol/Macro</th>
<th>Function/Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32-1+</td>
<td>Symbol Macro</td>
</tr>
<tr>
<td>int32-aref</td>
<td>Function</td>
</tr>
<tr>
<td>int32-logand</td>
<td>Function</td>
</tr>
<tr>
<td>int32-logandc1</td>
<td>Function</td>
</tr>
<tr>
<td>int32-logandc2</td>
<td>Function</td>
</tr>
<tr>
<td>int32-logbitp</td>
<td>Function</td>
</tr>
<tr>
<td>int32-logeqv</td>
<td>Function</td>
</tr>
<tr>
<td>int32-logior</td>
<td>Function</td>
</tr>
<tr>
<td>int32-lognand</td>
<td>Function</td>
</tr>
<tr>
<td>int32-lognot</td>
<td>Function</td>
</tr>
<tr>
<td>int32-logorc1</td>
<td>Function</td>
</tr>
<tr>
<td>int32-logorc2</td>
<td>Function</td>
</tr>
<tr>
<td>int32-logtest</td>
<td>Function</td>
</tr>
<tr>
<td>int32-logxor</td>
<td>Function</td>
</tr>
<tr>
<td>int32-minusp</td>
<td>Function</td>
</tr>
<tr>
<td>int32-plusp</td>
<td>Function</td>
</tr>
<tr>
<td>int32-to-int32</td>
<td>Function</td>
</tr>
<tr>
<td>int32-to-int64</td>
<td>Function</td>
</tr>
<tr>
<td>int32-to-integer</td>
<td>Function</td>
</tr>
<tr>
<td>int32-zerop</td>
<td>Function</td>
</tr>
<tr>
<td>int64+</td>
<td>Function</td>
</tr>
<tr>
<td>int64-</td>
<td>Function</td>
</tr>
<tr>
<td>int64-type</td>
<td>Function</td>
</tr>
<tr>
<td>int64/=</td>
<td>Function</td>
</tr>
<tr>
<td>int64/&lt;</td>
<td>Function</td>
</tr>
<tr>
<td>int64&lt;=</td>
<td>Function</td>
</tr>
<tr>
<td>int64=&gt;</td>
<td>Function</td>
</tr>
<tr>
<td>int64&gt;=</td>
<td>Function</td>
</tr>
<tr>
<td>+int64-0+</td>
<td>Symbol Macro</td>
</tr>
<tr>
<td>int64-1+</td>
<td>Function</td>
</tr>
<tr>
<td>int64-1-</td>
<td>Function</td>
</tr>
<tr>
<td>+int64-1+</td>
<td>Symbol Macro</td>
</tr>
<tr>
<td>int64-aref</td>
<td>Function</td>
</tr>
<tr>
<td>int64-logand</td>
<td>Function</td>
</tr>
<tr>
<td>int64-logandc1</td>
<td>Function</td>
</tr>
<tr>
<td>int64-logandc2</td>
<td>Function</td>
</tr>
<tr>
<td>int64-logbitp</td>
<td>Function</td>
</tr>
<tr>
<td>int64-logeqv</td>
<td>Function</td>
</tr>
<tr>
<td>int64-logior</td>
<td>Function</td>
</tr>
<tr>
<td>int64-lognand</td>
<td>Function</td>
</tr>
<tr>
<td>int64-lognot</td>
<td>Function</td>
</tr>
<tr>
<td>int64-logorc1</td>
<td>Function</td>
</tr>
<tr>
<td>int64-logorc2</td>
<td>Function</td>
</tr>
<tr>
<td>int64-logtest</td>
<td>Function</td>
</tr>
<tr>
<td>int64-logxor</td>
<td>Function</td>
</tr>
<tr>
<td>int64-minusp</td>
<td>Function</td>
</tr>
</tbody>
</table>

**Index**

- **interface**
  - between parser generator and lexical analyzer 317
  - Common SQL initialization 334
- **interactive-stream-p**
  - function 676

**interpreter**
- differences from compiler 101
- interruptable 105
- invalid superclass 253

- **:invisible**
  - keyword 762

**iOS interface**
- overview 241
- creating 241
- example 244
- IP Address 578

- **ip-address-string**
  - function 585
- IPv4 578
- IPv6 410, 578
  - on Mac OS X 410
  - on Windows 410
- IPv6 support 585, 600
- **ipv6-address**
  - function 586
- **ipv6-address-p**
  - function 587
- **ipv6-address-scope-id**
  - function 587
- **ipv6-address-string**
  - function 588
- ISO8859-1 443

**J**

- **jaref**
  - function 1260

**Java classes**
- **BugFormLogsList** 1357
- **BugFormViewer** 847
- **LispCalls** 1331
- **Manager** 1339

**Java fields**
- **mMaxErrorLogsNumber** 1350
- **mMessagesMaxLength** 1352

**Java interface**
Index

Java side 1331
Lisp side 197
Java interfaces
LispErrorReporter 1347
LispGuiErrorReporter 1347
MessageHandler 1354
Java methods
addMessage 1352
callDoubleA 1332
callDoubleV 1332
callIntA 1332
CallIntV 1332
callObjectA 1332
callObjectV 1332
callVoidA 1332
callVoidV 1332
checkLispSymbol 1334
clearBugFormLogs 1350
createLispProxy 1335
getApplicationContext 1355
getClassLoader 1355
init 1341
init_result_code 1344
loadLibrary 1346
mInitErrorString 1346
setCurrentActivity 1356
setErrorReporter 1347
setGuiErrorReporter 1347
setMessageHandler 1354
setTextView 1354
showBugFormLogs 1351
status 1343
Java virtual machine 201
java-array-element-type function 1262
java-array-error condition 1263
java-array-error-array accessor 1263
java-array-error-caller accessor 1263
java-array-indices-error condition 1263
java-array-indices-error-indices accessor 1264
java-array-indices-error-rank accessor 1264
java-array-length function 1264
java-array-simple-error condition 1265
java-bad-jobobject condition 1265
java-bad-jobobject-caller accessor 1265
java-bad-jobobject-object accessor 1265
Java-class-error condition 1266
java-definition-error condition 1266
java-definition-error-class-name accessor 1266
java-definition-error-name accessor 1266
java-exception condition 1267
java-exception-exception-name accessor 1268
java-exception-java-backtrace accessor 1268
java-exception-string accessor 1268
java-field-class-name-for-setting function 1302
java-field-error condition 1266
java-field-error-field-name accessor 1266
java-field-error-static-p accessor 1266
java-field-setting-error condition 1268
java-field-setting-error-class-name accessor 1269
java-field-setting-error-class-name-for-setting accessor 1269
java-field-setting-error-field-name accessor 1269
java-field-setting-error-field-name new-value accessor 1269
java-id-exception condition 1269
java-instance-jobobject accessor 219, 1316
java-instance-without-jobobject-error condition 1270
java-interface-error condition 1270
java-low-level-exception condition 1271
java-method-error condition 1266
java-method-error-args-num accessor 1266
java-method-error-method-name accessor 1266
java-method-exception condition 1271
java-method-exception-args accessor 1272
java-method-exception-class-name accessor 1272
java-method-exception-method-name accessor 1272
java-normal-exception condition 1272
java-not-a-java-object-error condition 1273
java-not-an-array-error condition 1274
*java-null* constant 1274
java-object-array-element-type function 1275
java-objects-eq function 1275
java-out-of-bounds-error condition 1276
java-primitive-array-element-type function 1277
java-serious-exception condition 1278
java-storing-wrong-type-error condition 1276
java-type-to-lisp-array-type function 1278
java-vm-poi type 1280
jboolean type 1280
jbyte type 1280
jchar type 1280
jdouble type 1280
jfloat type 1280
jint type 1280
jis 444
:jis external format 444
jlong type 1280
JNI_CreateJavaVM 1255, 1257
JNI_GetCreatedJavaVMS 1247
JNI_OnLoad 1257
jni-env-poi type 1281
:object initarg 1316
:object type 1281
:object-class-name function 1282
:object-ensure-global function 1283
:object-of-class-p function 1284
:object-p function 1285
:object-pretty-class-name function 1285
:object-string function 1286
:object-to-lisp function 1287
join slot 354
jshort type 1280
JVM 201
jvref function 1288

K

keywords
:after 50
:all 309
:allocation 55
:backtrace 52
:before 49
:bindings 757
:break 51
:break-on-exit 51
:catchers 760
:caused-by 309
:default-pathname 307
:dont-know 17
:entrycond 52
:eval-after 51
:eval-before 50
:exitcond 52
:handler 761
:hidden 743
:inside 55
:invisible 762
:maximum-overflow 143
:members 307
:minum-for-sweep 139, 143
:minum-overflow 143
:new-generation-size 144
:package 307
:previous 309
:process 54
:requires 309, 310
:restarts 764
:rules 308
:source-only 308
:step 52
:trace-output 54
:when 55
KnowledgeWorks rules
compiling dynamically 103
known-sid-integer-to-sid-string function 1901
KOI8 445
:koi-8 external format 445
KOI8-R 445

L

:1 debugger command 17
Index

:lambda debugger command 19
last-callback-on-thread function 1399
Latin-1 443
\*latin-1\* external format 443
*latin-1-code-pages* variable 1902
\*latin-1-safe* external format 444
\*latin-1-terminal* external format 443
levels of safety, see compiler
:if debugger command 19
library formats 1079
libssl32.dll OpenSSL DLL 420
*lisp-arguments-list* variable 454, 1795
lisp image
filename 454, 1149
pathname 454, 1149
lisp-array-to-primitive-array function 1299
lisp-array-type-to-java-type function 1278
LispCalls Java class 1331
LispGuiErrorReporter Java interface 1347
LispImageErrorReporter Java interface 1347
lisp-image-name function 454, 1149
lisp-java-instance-p function 1290
lisp-name-to-foreign-name function 1078
lisp-to-jobobject function 1290
LispWorks customizing 178
processes 261
quitting 4, 99
saving 2
starting 1, 99
threads in 261
LispWorks as a DLL 191
LispWorks as a dynamic library 191
LispWorks as a shared library 191
LispWorks IDE
Debugger tool 1124
Help menu lviii, 249, 639
Inspector tool 1149
Notifier window 1124
Profiler tool 999
*lispworks-directory* variable 1150
LispWorksDlsym C function 1988
LispWorksState C function 194, 1989
List Callers Editor command 863
list-all-processes function 263, 1400
list-attributes function 353, 1574
list-attribute-types function 353, 1573
list-classes function 357, 1575
listener 1181
main chapter 5
top level commands 1740
listener commands
?:7
:bug-form 7
:get 6
:help 7
:his 7
:redo 6
:use 7
listener process 1398
listener prompt 1163
list-sql-streams function 372, 1576
list-tables function 353, 1576
Load File Editor command 120
load-all-patches function 1152
load-code-coverage-data function 889
load-data-file function 957
*load-fasl-or-lisp-file* variable 961
LoadLibrary C function 191
loadLibrary Java method 1346
load-logical-pathname-transformations function 677
load-on-demand 188
*load-source-if-newer* 1154
load-system function 1153
:lob-locator initarg 1577
lob-stream class 380, 384, 389, 1577
lob-stream-lob-locator accessor 1578
lob-stream-lob-locator function 384
local-dspec-p function 812
locale-file-encoding function 1796
locally-disable-sql-reader-syntax function 370, 1578
locally-enable-sql-reader-syntax function 370, 1579
location macro 813
lock system class 1401
lock-and-condition-variable-
broadcast function 1401
lock-and-condition-variable-
signal function 1402
lock-and-condition-variable-
wait function 1404
lock-locked-p function 1406
lock-name function 277, 1409
lock-owned-by-current-pro-
cess-p function 1407
lock-owner function 277, 1409
lock-recursively-locked-p function 1408
lock-recursive-p function 1407
locks 277
log-bug-form function 754
logs-directory function 755
long-float type 678
long-namestring function 454, 1903, 1915
long-site-name function 453, 678
loop macro 346, 351, 679, 1580
extensions in Common SQL 358
loop, extensions in Common SQL 351
loop-processing-wait-state-
collection function 411, 413, 589
Low level atomic operations 299
low-level-atomic-place-p function 1797
lpstr FLI type 1916
lpctstr FLI type 1917
lpowstr FLI type 1918
lptstr FLI type 1916
lpwstr FLI type 1917
lpwstr FLI type 1918
M
:mini inspector command 39, 40
Mach-O bundle 1005
Mach-O dynamically linked shared library 1003
:macos-roman external format 444
macros advice 67
allocation-in-gen-num 137, 146, 843
allowing-block-interrupts 289, 1359
analysing-special-variables-usage 844
appendf 1083
at-location 792
atomic-decf 1718
atomic-exchange 1719
atomic-fixnum-decf 1719
atomic-fixnum-incf 1719
atomic-incf 1718
atomic-pop 1720
atomic-push 1721
block-promotion 861
catching-exceptions-bind 1211
catching-java-exceptions 1211
cd 864
catch-and-swap 1733
declai 110, 650
def 794
defadvice 67, 73, 1100
defclass 657
defglobal-parameter 903
defglobal-variable 904
define-action 96, 1105
define-action-list 96, 1107
define-atomic-modify-
macro 1738
define-dde-client 1967
define-dde-desh-patch-
topic 1977
define-dde-server 1978
define-dde-server-
function 1979
define-declaration 905
define-dspec-alias 795
define-dspec-class 796
define-field-accessor 1219
define-foreign-callable 297
define-form-parser 799
define-java-caller 1220
define-java-callers 1223
define-java-constructor 1220
define-lisp-proxy 1225
define-top-loop-command 1740
defpackage 661
defparameter 904
defparser 315, 1515
defstruct 189
defsystem 305–311, 1109
deftransform 725
defvar 904
def-view-class 354, 355, 1551
Index

delete-advice 66, 73, 907

do-query 351, 381, 1563

error-situation-forms 923

execute-actions 1129

extended-time 149, 163, 173, 926

globally-accessible 269, 270, 273, 274, 1768

impersonating-named-pipe-client 1899

impersonating-user 1900

import-java-class-definitions 1251

in-static-area 137, 1771

location 813

loop 679, 1580

profile 169, 981

push-end 1164

push-end-new 1164

rebinding 1167

release-object-and-nullify 1837

remove 1171

replacement-source-form 820

restart-case 694

simple-do-query 351, 381, 1640

step 706

throw-if-tag-found 1040

time 711

trace 713

undefine-action 96, 1187

undefine-action-list 96, 1188

untrace 721

unwind-protect-blocking-interrupts 289, 1055

unwind-protect-blocking-interrupts-in-cleanups 289, 1057

when-let 1201

when-let* 1202

with-action-item-error-handling 1204

with-action-list-mapping 1205

with-code-coverage-generation 1062

with-dde-conversation 1968

with-debugger-wrapper 785

with-ensuring-gethash 1063

with-exclusive-lock 277, 1509

with-hash-table-locked 1065

with-heavy-allocation 146, 1066

with-interrupts-blocked 289, 1510

with-lock 277, 1510

with-modification-change 1890

with-modification-check-macro 1891

with-noticed-socket-stream 637

with-other-threads-disabled 289, 1892

without-code-coverage 1071

without-interrupts 289, 290, 1512

without-preemption 289, 290, 1513

without-output-to-fasl-file 1067

with-output-to-string 723

with-registry-key 468, 1938

with-remote-debugging-connection 788

with-remote-debugging-spec 789

with-ring-locked 1071

with-sharing-lock 277, 1511

with-sqlite-blob 395, 1677

with-stream-input-buffer 1710

with-stream-output-buffer 1712

with-transaction 348, 350, 1678

with-unique-names 1206

with-windows-event-log-event-source 1921

mailbox system class 1411

mailbox-count function 1411

mailbox-empty-p function 1412

mailbox-full-p function 1413

mailbox-not-empty-p function 1414

mailbox-peek function 1414

mailbox-read function 1415

mailbox-reader-process function 1416

mailbox-send function 1417

mailbox-send-limited function 1418

mailbox-size function 1420

mailbox-wait function 1421

mailbox-wait-for-event function 1422
Index

*main-process* variable 1424
make-array function 165, 276, 680
make-barrier function 287, 1424
make-condition-variable function 1426
make-current-allocation-permanent function 1798
make-gesture-spec function 1800
make-hash-table function 165, 276, 682
make-instance generic function 684
make-java-array function 1291
make-java-instance function 1292
make-lisp-proxy function 1293
make-lisp-proxy-with-overrides function 1293
make-lock function 1427
make-mailbox function 1429
make-method-lambda generic function 250
make-mt-random-state function 1156
make-named-timer function 1430
make-object-permanent function 1805
make-permanent-simple-vector function 1807
make-ring function 962
make-semaphore function 288, 1431
make-sequence function 685
make-simple-int32-vector function 1808
make-simple-int64-vector function 1809
make-ssl-ctx function 422, 423, 588
make-stderr-stream function 1810
make-string function 686
make-string-output-stream function 687
make-symbol function 164
make-timer function 293, 1432
make-typed-aref-vector function 1811
make-unlocked-queue function 964
make-unregistered-action-list function 1154
make-wait-state-collection function 411, 590
Manager Java class 1339
map function 688
map-all-processes function 1433
map-all-processes-backtrace function 1433
map-code-coverage-data function 966
map-environment function 1812
map-java-object-array function 1295
map-process-backtrace function 1434
map-processes function 1435
map-query function 351, 381, 1582
map-ring function 967
mark and sweep 138
mark-and-sweep function 967
marking-gc function 150, 1813
*maximum-ordinary-windows* variable 186
:maximum-overflow keyword 143
*max-trace-indent* variable 59, 969
:members keyword 307
memory allocation during tracing 55
memory clashes 194
avoiding 459
memory management common features 163
garbage collection strategy in 32-bit LispWorks 139
image reduction 163
image relocation 457
in 32-bit LispWorks 135
in 64-bit LispWorks 146
main chapter 129
mark and sweep 138
overflow 143
relocating the image 457
timing in 133, 149, 163
memory-growth-margin function 146, 1815
merge function 688
merge-code-coverage-data function 970
merge-e1-specs function 1816
Mersenne Twister 1157
MessageHandler Java interface 1354
Metaobject Protocol 249
metaobject protocol class options 495
slot options 498
method advice 67
method-combination class 251
Index

methods
  tracing 57
  waitForInitialization 1336
:minimum-for-sweep keyword 139, 143
:minimum-overflow keyword 143
mInitErrorString Java method 1346
mMaxErrorLogsNumber Java field 1350
mMessagesMaxLength Java field 1352
mobile-gc-p function 1817
mobile-gc-sweep-objects function 1818
mod 2^32 arithmetic 472
mod 2^64 arithmetic 474
modify-hash function 266, 267, 971
modifying a database 348
MOP
  AMOP compatibility 249
  class options 495
  slot options 498
most-positive-fixnum constant 479
mt-random function 1157
mt-random-state type 1158
*mt-random-state* variable 1157
*mt-random-state-p* function 1158
*multibyte-code-page-ef* variable 391, 1904
multi-processing
  locks 277
multiprocessing
  locks 277
:multithreading delivery keyword 264
MySQL
  connecting 339
MySQL client library 341
  Mac OS X 341
*mysql-library-directories* variable 341, 342, 1583
*mysql-library-path* variable 341, 342, 1584
*mysql-library-sub-directories* variable 341, 1585
N
  :n debugger command 16
name-defined-dspecs function 813
name-definition-locations function 814
named-pipe-stream-name function 1904
name-only-form-parser function 815
New in LispWorks 7.0
  accepting-handle type 517
  accepting-handle-collection function 518
  accepting-handle-local-port function 520
  accepting-handle-name function 519
  accepting-handle-socket function 520
  accepting-handle-user-info function 521
  accept-tcp-connections-creating-async-io-states function 513
  add-code-coverage-data function 839
addMessage Java method 1352
  AIX PowerPC/32-bit LispWorks 649
  AIX PowerPC/64-bit LispWorks 649
:android-delivery LispWorks 671
android-funcall-in-main-thread function 849
android-get-current-activity function 851
*android-main-process-for-testing* variable 852
android-main-thread-p function 852
apply-in-wait-state-collection-process function 522
approaching-memory-limit condition 1717
:arm feature 671
ARM Linux/32-bit LispWorks 649
Asynchronous I/O 411
async-io-state structure type 523
async-io-state-abort function 523
async-io-state-abort-and-close function 525
async-io-state-address function 526
async-io-state-buffered-data-length function 528
async-io-state-discard function 531
async-io-state-finish
  function 532
async-io-state-get-bufferred-data
  function 533
async-io-state-peer-address
  function 538
async-io-state-read-buffer
  function 538
async-io-state-read-with-checking
  function 542
async-io-state-receive-message
  function 545
async-io-state-send-message
  function 547
async-io-state-send-message-to-address
  function 549
async-io-state-write-buffer
  function 552
*background-input* variable 858
*background-output* variable 858
*background-query-collection* variable 858
backlog argument to accept-tcp-connections-creating-async-io-states 514
backlog argument to start-up-server 626
base-char-ref function 1821
:bmp external format 827
bmp-char type 1086
bmp-char-p function 1087
:bmp-native external format 827
:bmp-reversed external format 827
bmp-string type 1088
bmp-string-p function 1089
brackets-limits argument to find-regexp-in-string 1135
BugFormLogsList Java class 1357
BugFormViewer Java class 1357
callDoubleA Java method 1332
callDoubleV Java method 1332
callIntA Java method 1332
callIntV Java method 1332
call-java-method function 1209
call-java-method-error
  condition 1210
callObjectA Java method 1332
callObjectV Java method 1332
callVoidA Java method 1332
callVoidV Java method 1332
call-wait-state-collection
  function 556
catching-exceptions-bind macro 1211
catching-java-exceptions macro 1211
checked-read-java-field function 1302
check-java-field function 1302
check-lisp-calls-initialized function 1213
checkLispSymbol Java method 1334
clearBugFormLogs Java method 1350
clear-code-coverage function 869
close-accepting-handle function 557
close-async-io-state function 558
close-wait-state-collection function 559
code-coverage-data type 870
code-coverage-data-create-time accessor 870
code-coverage-data-generate-coloring-html function 871
code-coverage-data-generate-statistics function 877
code-coverage-data-name accessor 870
code-coverage-file-stats
  type 879
code-coverage-file-stats-called accessor 879
code-coverage-file-stats-counters-count accessor 879
code-coverage-file-stats-counters-executed accessor 879
code-coverage-file-stats-counters-hidden accessor 879
code-coverage-file-stats-fully-covered accessor 879
code-coverage-file-stats-hidden-covered accessor 879
code-coverage-file-stats-lambdas-count accessor 879
code-coverage-file-stats-not-called accessor 879
code-coverage-file-stats-
Index

partially-covered accessor 879
code-coverage-file-stats-source-file accessor 879
code-coverage-set-editor-colors function 881
code-coverage-set-editor-default-data function 883
code-coverage-set-html-background-colors function 884
compute-effective-method-function-from-classes generic function 490
Conditional throw and checking for catch in the dynamic environment 476
copy-code-coverage-data function 889
copy-current-code-coverage function 889
copy-standard-object function 493
create-and-run-wait-state-collection function 560
create-async-io-state function 562
create-async-io-state-and-connected-tcp-socket function 564
create-async-io-state-and-connected-udp-socket function 567
create-async-io-state-and-udp-socket function 569
create-instance-from-job-ject function 1213
create-instance-jobobject function 1214
create-instance-jobobject-list function 1214
create-java-object function 1216
create-java-object-error condition 1217
createLispProxy Java method 1335
current-process-kill function 1382
current-process-send function 1385
current-process-set-terminating-method function 1386
decode-to-db-standard-date function 1547
decode-to-db-standard-timestamp function 1547
default-constructor-arguments generic function 1217
default-name-constructor function 1218
define-field-accessor macro 1219
define-java-caller macro 1220
define-java-callers macro 1223
define-java-constructor macro 1220
define-lisp-proxy macro 1225
delivered-image-p function 908
deliver-to-android-project function 909
destructive-add-code-coverage-data function 839
destructive-merge-code-coverage-data function 970
destructive-reverse-subtract-code-coverage-data function 839
destructive-subtract-code-coverage-data function 839
detect-utf32-bom function 1744
detect-utf8-bom function 1744
dump-forms-to-file additional arguments and improved documentation 1067
editor-color-code-coverage function 917
:embedded-module member option for defsystem :type :c-file 1112
encode-db-standard-date function 1548
encode-db-standard-timestamp function 1548
ensure-hash-entry function 922
ensure-is-jobobject function 1247
ensure-lisp-classes-from-tree function 1233
ensure-supers-contain-java.lang.object function 1235
:error argument in gesture-spec-to-character 1760
errorp argument to gesture-spec-to-character 1760
error-situation-forms macro 923
example-edit-file function 1128
fasl-error condition 929
fast-directory-files function 930
fdf-handle-directory-p accessor 930
fdf-handle-directory-string accessor 930
fdf-handle-last-access accessor 930
fdf-handle-last-modify accessor 930
fdf-handle-link-p accessor 930
fdf-handle-size accessor 930
fdf-handle-writable-p accessor 930
field-access-exception condition 1236
field-access-exception-set-p accessor 1236
field-exception condition 1237
field-exception-class-name accessor 1237
field-exception-field-name accessor 1237
filter-code-coverage-data function 933
find-encoding-option supports GNU Emacs coding option 1752
find-java-class function 1238
find-throw-tag function 936
format-to-java-host function 1239
funcall-async function 1392
funcall-async-list function 1392
generate-code-coverage function 946
generate-java-class-definitions function 1239
gesture-spec-caps-lock-bit constant 1756
getApplicationContext Java method 1355
getClassLoader Java method 1355
get-code-coverage-delta function 940
get-default-local-ipv6-address function 578
get-folder-path is now available on all platforms 1762
get-gc-timing function 1031
gethash-ensuring function 952
get-ip-default-zone-id function 581
get-java-virtual-machine function 1246
get-jobject function 1247
get-primitive-array-region function 1248
get-service-entry function 582
get-superclass-and-interfaces-tree function 1250
immediatep function 1770
import-java-class-definitions macro 1251
init Java method 1341
init_result_code Java method 1344
int32-to-int64 function 1781
int64* function 1784
int64+ function 1784
int64- function 1784
int64-type 1783
int64/= function 1784
int64/= function 1786
int64< function 1786
int64<< function 1787
int64<< function 1786
int64= function 1786
int64>= function 1786
int64>> function 1787
int64>>>= function 1787
int64-0+ symbol macro 1785
int64-0+ symbol macro 1785
int64-1+ function 1785
int64-1- function 1785
+int64-0+ symbol macro 1785
int64-aref function 1788
int64-logand function 1789
int64-logandcl function 1789
int64-logandc2 function 1789
int64-logbitp function 1789
int64-logeqv function 1789
int64-logior function 1789
int64-logging function 1789
int64-lognot function 1789
int64-logorcl function 1789
int64-logorc2 function 1789
int64-logtest function 1789
Index

int64-logxor function 1789
int64-plusp function 1791
int64-to-int32 function 1792
int64-to-integer function 1793
int64-zerop function 1791
integer-to-int64 function 1794
intern-and-export-list
  function 1260
:ios-delivery feature 671
jaref function 1260
java-array-element-type
  function 1262
java-array-error condition 1263
java-array-error-array
  accessor 1263
java-array-error-caller
  accessor 1263
java-array-indices-error
  condition 1263
java-array-indices-error-indices accessor 1264
java-array-indices-error-rank accessor 1264
java-array-length function 1264
java-array-simple-error
  condition 1265
java-bad-jobject condition 1265
java-bad-jobject-caller accessor 1265
java-bad-jobject-object
  accessor 1265
java-class-error condition 1266
java-definition-error
  condition 1266
java-definition-error-class-name accessor 1266
java-definition-error-name
  accessor 1266
java-exception condition 1267
java-exception-exception-name accessor 1268
java-exception-java-backtrace accessor 1268
java-exception-string accessor 1268
java-field-class-name-for-setting function 1302
java-field-error condition 1266
java-field-error-field-name
  accessor 1266
java-field-error-static-p
  accessor 1266
java-field-setting-error
  condition 1268
java-field-setting-error-class-name accessor 1269
java-field-setting-error-class-name-for-setting
  accessor 1269
java-field-setting-error-field-name accessor 1269
java-field-setting-error-field-name
  accessor 1269
java-instance-jobject
  accessor 1316
java-instance-without-jobject-error condition 1270
java-interface-error
  condition 1270
java-low-level-exception
  condition 1271
java-method-error
  condition 1266
java-method-error-args-num accessor 1266
java-method-error-method-name accessor 1266
java-method-exception
  condition 1271
java-method-exception-args
  accessor 1272
java-method-exception-class-name accessor 1272
java-method-exception-method-name accessor 1272
java-method-exception-name accessor 1272
java-normal-exception
  condition 1272
java-not-a-java-object-error condition 1273
java-not-an-array-error
  condition 1274
java-object-array-element-type function 1275
java-objects-eq function 1275
java-out-of-bounds-error
  condition 1276
java-primitive-array-element-type function 1277
java-serious-exception
  condition 1278
Index

java-storing-wrong-type-error condition 1276
java-type-to-lisp-array-type function 1278
java-vm-poi FLI type
descriptor 1280
jboolean FLI type descriptor 1280
jbyte FLI type descriptor 1280
jchar FLI type descriptor 1280
jdouble FLI type descriptor 1280
jfloat FLI type descriptor 1280
jint FLI type descriptor 1280
jlong FLI type descriptor 1280
jni-env-poi FLI type
descriptor 1281
jobject FLI type descriptor 1281
jobject-class-name function 1282
jobject-ensure-global function 1283
jobject-of-class-p function 1284
jobject-p function 1285
jobject-pretty-class-name function 1285
jobject-string function 1286
jobject-to-lisp function 1287
jshort FLI type descriptor 1280
jvref function 1288
KOI8-R external format 445
lisp-array-to-primitive-array function 1299
lisp-array-type-to-java-type function 1278
LispCalls Java class 1331
LispErrorReporter Java interface 1347
LispGuiErrorReporter Java interface 1347
lisp-java-instance-p function 1290
lisp-to-jobobject function 1290
LispWorks for Android Runtime 649
LispWorks for iOS Runtime 649
load-code-coverage-data function 889
loadLibrary Java method 1346
loop-processing-wait-state-collection function 589
mailbox-wait function 1421
make-java-array function 1291
make-java-instance function 1292
make-lisp-proxy function 1293
make-lisp-proxy-with-overrides function 1293
make-ring function 962
make-simple-int64-vector function 1809
make-unlocked-queue function 964
make-wait-state-collection function 590
Manager Java class 1339
map-java-object-array function 1295
map-ring function 966, 967
merge-code-coverage-data function 970
MessageHandler Java interface 1354
mInitErrorString Java method 1346
mMaxErrorLogsNumber Java field 1350
mMessagesMaxLength Java field 1352
object-pointer function 1820
octet-ref function 1821
open-tcp-stream-using-java function 595
Package change and additional functionality for load-data-file 957
package-flagged-p function 1827
pipe-exit-status function 1829
position-in-ring function 976
position-in-ring-forward function 976
primitive-array-to-lisp-array function 1299
process-interrupt-list function 1446
process-terminate function 1468
read-java-field function 1302
record-java-class-lisp-sym-
bol function 1304
record-message-in-windows-
event-log function 1909
replacement-source-form macro 820
replace-socket-stream-
socket function 664
replace-standard-object function 499
report-error-to-java-host
reset-code-coverage  function 869
reset-code-coverage-snapshot  function 940
reset-java-interface-for-new-jvm  function 1306
reset-ring  function 990
restore-code-coverage-data  function 869
reverse-subtract-code-coverage-data  function 839
ring-length  function 991
ring-name  function 992
ringp  function 991
ring-pop  function 992
ring-push  function 993
ring-ref  function 994
rotate-ring  function 995
run-shell-command  is implemented on Windows 1839
save-code-coverage-data  function 889
save-current-code-coverage  function 889
save-exit-status  argument to open-pipe 1823
send-message-to-java-host  function 1307
sequencep  function 1174
set-approaching-memory-limit-callback  function 1845
set-code-coverage-snapshot  function 940
setCurrentActivity  Java  method 1356
setErrorReporter  Java  method 1347
setMessageHandler  Java  method 1354
set-primitive-array-region  function 1248
setTextView  Java  method 1354
setup-field-accessor  function 1312
setup-java-caller  function 1313
setup-java-constructor  function 1313
setup-lisp-proxy  function 1315
showBugFormLogs  Java  method 1351
simple-bmp-string  type 1088
simple-bmp-string-p  function 1089
simple-int64-vector  type 1880
socket-stream-shutdown  function 620
standard-java-object  class 1315
start-gc-timing  function 1031
status  Java  method 1343
stop-gc-timing  function 1031
string-append*  function 1183
structurep  function 1184
subtract-code-coverage-data  function 839
suspend  argument to stop-profiling 1036
switch-open-tcp-stream-with-ssl-to-java  function 634
symbol-dynamically-bound-p  function 1039
Test for dynamic bindings 476
The HTML version of the CLOS Metaobject protocol is available via the Help menu. 249
throw-an-exception  function 1317
throw-if-tag-found  macro 1040
unlocked-queue  type 1055
unlocked-queue-read  function 964
unlocked-queue-ready  function 964
unlocked-queue-send  function 964
User Guide chapter "Code Coverage" 119
User Guide section "Code signing in saved images" 181
User Guide section "Specifying the target for connecting and binding a socket" 408
value :delete  for keyword argument :load  of compile-file 645
value :none  for direction argument to open-pipe 1823
verify-java-caller
Index

function 1418
mailbox-size function 1420
make-current-allocation-
permanent function 1798
make-object-permanent
function 1805
make-permanent-simple-vec-
tor function 1807
map-code-coverage-data
function 966
mobile-gc-p function 1817
mobile-gc-sweep-objects
function 1818
object-dspec function 816
pipe-close-connection
function 1828
release-object-and-nullify
macro 1837
remote-debugging-connec-
tion system class 765
remote-debugging-connec-
tion-add-close-cleanup
function 765
remote-debugging-connec-
tion-name function 767
remote-debugging-connec-
tion-peer-address
function 768
remote-debugging-connec-
tion-remove-close-
cleanup function 765
remote-debugging-stream-
peer-address generic
function 769
remote-inspect function 770
remote-object-connection
function 772
remote-object-p function 772
replace-from-sqlite-blob
function 1650
replace-from-sqlite-raw-
blob function 1655
replace-into-sqlite-blob
function 1650
rotate-byte function 1172
safe-format-to-limited-
string function 996
safe-format-to-string
function 996
safe-prin1-to-string
function 996
safe-princ-to-string
function 996
save-current-profile-tree
function 998
set-default-remote-debug-
ging-connection function 774
set-expected-allocation-in-
generation-2-after-gc
function 1853
set-generation-2-gc-options
function 1858
set-process-profiling pro-
cesses value :new 1021
set-promote-generation-1
function 1868
set-remote-debugging-con-
nection function 775
set-reserved-memory-policy
function 1869
set-split-promotion
function 1872
set-static-segment-size
function 1874
setup-deliver-dynamic-
library-for-java
function 1308
setup-java-interface-call-
backs function 1253
simple-int32-vector-length
function 1879
simple-int32-vector-p
function 1879
simple-int64-vector-length
function 1880
simple-int64-vector-p
function 1881
socket-stream-handshake
function 618
software-version detects Win-
dows 10 correctly 704
specific-valid-file-encod-
ings* variable 1881
sql-expression-object system
class 1647
sqlite-blob system class 1648
sqlite-blob-length
function 1650
sqlite-blob-p function 1650
sqlite-close-blob function 1650
sqlite-last-insert-rowid
function 1649
sqlite-open-blob function 1650
sqlite-open-blob system class 1654
sqlite-open-blob-length function 1655
sqlite-open-blob-p function 1655
sqlite-open-blob-ref function 1655
sqlite-open-blob-valid-p function 1655
sqlite-reopen-blob function 1650
start-client-remote-debugging-server function 776
start-ide-remote-debugging-server function 779
start-remote-listener function 781
string-needs-n-prefix function 1668
string-prefix-with-n-if-needed function 1669
string-trim-whitespace function 1035
*to-java-host-stream* variable 1318
*to-java-host-stream-no-scroll* variable 1319
undefine-declaration function 1054
unlocked-queue-count function 964
unlocked-queue-peek function 964
unlocked-queue-size function 964
*use-n-syntax-for-non-ascii-strings* variable 1676
Using Asynchronous I/O with SSL 421, 425
waitForInitialization method 1336
with-remote-debugging-connection macro 788
with-remote-debugging-spec macro 789
with-sqlite-blob macro 1677
write-string-with-properties macro 1072
:new-generation-size keyword 144
Newly documented in LispWorks 7.0

sort-inspector-p generic function 1138
Newly documented in LispWorks 7.1
prepared-statement type 1629
push-end macro 1164
push-end-new macro 1164
non-terminal in grammar 314
normal-gc function 146, 972
notice-fd function 1436

O
object
object-oriented interface in Common SQL 354
static 137
object finalization 164
Object Oriented DDL in Common SQL 355
Object Oriented DML in Common SQL 357
object-address function 1819
object-dspec function 816
object-pointer function 1820
octet-ref function 1821
ODBC
connecting 339
OODDL 355
OODML 357
open function 689
opening a URL 1827
open-named-pipe-stream function 1905
open-pipe function 1823
open-registry-key function 468, 1932
open-serial-port function 1517
OpenSSL 418
OpenSSL DLLs
libcrypto-1_1.dll 420
libcrypto-1_1-x64.dll 420
libeay32.dll 420
libssl-1_1.dll 420
libssl-1_1-x64.dll 420
libssl32.dll 420
openssl-version function 599
open-stream-p generic function 691
open-tcp-stream function 420, 422, 591
open-tcp-stream-using-java function 595
open-temp-file function 467, 895
open-url function 1827
operating system 453
optimization
fast 32-bit arithmetic 472
fast 64-bit arithmetic 474
floating point 112
foreign slot access 116
of compiler 104
tail call 113, 173
optimization declarations 105
optimization hints 111, 651
optimize 105
optimize qualities 105–108
:optimize-slot-access class
option 250, 253, 504, 658
Oracle
connecting 338
Oracle Call Interface
in Common SQL 338
Oracle large objects 380
Oracle LOB interface 380
Oracle locator objects 380
ora-lob-append function 388, 1586
ora-lob-assign function 387, 1587
ora-lob-char-set-form function 386, 1587
ora-lob-char-set-id function 1588
ora-lob-close function 388, 1589
ora-lob-copy function 388, 1590
ora-lob-create-empty function 381, 387, 1591
ora-lob-create-temporary function 389, 1592
ora-lob-disable-buffering function 389, 1593
ora-lob-element-type function 386, 1594
ora-lob-enable-buffering function 389, 1595
ora-lob-env-handle function 384, 1596
ora-lob-erase function 388, 1596
ora-lob-file-close function 388, 1597
ora-lob-file-close-all function 388, 1598
ora-lob-file-exists function 1599
ora-lob-file-get-name function 1600
ora-lob-file-is-open function 1600
ora-lob-file-open function 388, 1601
ora-lob-file-set-name function 388, 1602
ora-lob-flush-buffer function 389, 1603
ora-lob-free function 387, 1604
ora-lob-free-temporary function 389, 1605
ora-lob-get-buffer function 384, 389, 1605
ora-lob-get-chunk-size function 387, 1608
ora-lob-get-length function 387, 1608
ora-lob-internal-lob-p function 386, 387, 1609
ora-lob-is-equal function 387, 1610
ora-lob-is-open function 387, 1610
ora-lob-is-temporary function 387, 389, 1611
ora-lob-load-from-file function 388, 1612
ora-lob-lb-locator function 384, 1613
ora-lob-locator-is-init function 387, 1614
ora-lob-open function 388, 1614
ora-lob-read-buffer function 386, 389, 1615
ora-lob-read-foreign-buffer function 384, 386, 389, 1619
ora-lob-read-into-plain-file function 389, 1618
ora-lob-svc-ctx-handle function 384, 1620
ora-lob-trim function 388, 1620
ora-lob-write-buffer function 386, 389, 1621, 1624
ora-lob-write-foreign-buffer function 384, 386, 389
ora-lob-write-from-plain-file function 389
ora-lob-write-from-plain-file function 1623
output trace 54
output-backtrace function 756
output-stream-p generic function 692
P
:p debugger command 15
package
Index

hiding 24
:package keyword 307
package-flagged-p function 1827
Packages
allocation of 164
*packages-for-warn-on-redefinition* variable 187, 973
parameters
command line 454
*default-character-element-type* 438, 440, 441, 442, 449, 1104
parse-float function 975
parse-form-dspec function 817
parser generator main chapter 313
parser, error handling 316
passing run time parameters 454
patches
saving an image with 4
pathname comparison
case-sensitivity on Mac OS X 469
pathname of deliverable 454, 1149
pathname of DLL 454, 1149
pathname of dynamic library 1149
pathname of executable 454, 1149
pathname of dynamic library 1149
pathname of Lisp image 454, 1149
pathname-location function 1159
pem-read function 426
pem-read function 602
pipe
open 1823
pipe-close-connection function 1828
pipe-exit-status function 1829
pipe-kill-process function 1830
PL/SQL 1567
platform 453
*features* 670
software-type 703
software-version 704
p-oci-env FLI type 385, 1625
p-oci-file FLI type 385, 1626
p-oci-lob-locator FLI type 385, 1626
p-oci-lob-or-file FLI type 1626
p-oci-svc-ctx FLI type 385, 1626
pointer-from-address function 1832
pointers
weak 1012
position-in-ring function 976
position-in-ring-forward function 976
PostgreSQL
connecting 342
PostScript Printer Description files 189
PPD files 189–190
precompile-regexp function 1160
prepared-statement type 1629
prepare-statement function 1627
:previous keyword 309
primitive-array-to-lisp-array function 1299
print-action-lists function 97, 1161
print-actions function 97, 1161
*print-binding-frames* variable 24, 757
*print-catch-frames* variable 24, 759
*print-command* variable 1162
printer
configuring 189
*print-handler-frames* variable 24, 760
*print-invisible-frames* variable 762
*print-nickname* variable 1162
print-object generic function 189
*print-open-frames* variable 763
*print-pretty-gesture-spec function 1833
print-profile-list function 171, 977
print-query function 348, 1629
*print-restart-frames* variable 24, 764
*print-string* variable 981
*print-symbols-using-bars* variable 1834
process
creation 262
current 262
in LispWorks 261
scheduling 295
process exit status 463
:process keyword 54
process plist 294
process properties 294
process waiting 280
process-a-class-option generic function 495
process-alive-p function 1436
process-all-events function 1437
process-allow-scheduling
Index

property function 294, 1483
pushnew-to-process-property
function 294, 1484

Q
:q inspector command 39
query function 352, 380, 1631
*query-io* variable 705
query-registry-key-info
function 468, 1933
query-registry-value
function 468, 1934
quick backtrace 14
quit function 4, 463, 1165
QuitLispWorks C function 195, 1991
quitting LispWorks 4, 99 :
:quit-when-no-windows delivery
keyword 1176

R
raw 32-bit arithmetic 472
raw 64-bit arithmetic 474
read-dhparams function 426, 603
read-eval-print loop 5, 1181
read-foreign-modules
function 1078
read-java-field function 1302
read-serial-port-char
function 1521
read-serial-port-string
function 1521
:read-timeout initarg 613
real time 169
rebinding macro 1167
reconnect function 338, 1632
record-definition function 817
record-java-class-lisp-sym-
bol function 1304
record-message-in-windows-
event-log function 1909
*record-source-files* variable 818
*redefinition-action* variable 187, 819
redo 188
:redo listener command 6
reduce-memory function 164, 985
references-who function 987
regexp 1135, 1168
regexp-find-symbols
function 1168
registry
API on Windows 468, 1923
registry-key-exists-p
function 468, 1935
registry-value function 468, 1935
regular expression 1135, 1168
regular expression matching 1116, 1135,
1168
release-object-and-nullify
macro 1837
relocating 459
remote debugging 25
remote-debugging-connection
system class 765
remote-debugging-connection-
add-close-cleanup
function 36, 765
remote-debugging-connection-
name function 36, 767
remote-debugging-connection-
peer-address function 36,
768
remote-debugging-connection-
remove-close-cleanup
function 36, 765
remote-debugging-stream-
peer-address generic
function 769
remote-inspect function 25, 27, 28,
34, 770
remote-object-connection
function 29, 35, 772
remote-object-p function 29, 772
remove-advice function 66, 73, 1169
remove-duplicates function 116
remove macro 1171
remove-from-process-private-
property function 294, 1486
remove-from-process-property
function 294, 1486
remove-process-private-prop-
erty function 294, 1487
remove-process-property
function 294, 1488
remove-special-free-action
function 165, 988
remove-symbol-profiler
function 989
REPL 5, 1181
REPL inspector 37
replace-from-sqlite-blob
function 395, 1650
replace-from-sqlite-raw-blob
function 395, 1655
replace-into-sqlite-blob
function 395, 1650
replacement-source-form
macro 820
replace-socket-stream-socket
function 604
replace-standard-object
function 499
report-error-to-java-host
function 1305
:requires keyword 309, 310
*require-verbose* variable 1171
:res debugger command 21
reserved words 973
reset-code-coverage function 869
reset-code-coverage-snapshot
function 940
reset-java-interface-for-
new-jvm function 1306
reset-profiler function 989
reset-ring function 990
restart 12
restart frame, examining 12
:restart-case macro 694
:restarts keyword 764
restore-code-coverage-data
function 869
restore-sql-reader-syntax-
state function 370, 1633
:ret debugger command 21
reverse-subtract-code-cover-
age-data function 839
ring-length function 991
ring-name function 992
ringp function 991
ring-pop function 992
ring-push function 993
ring-ref function 994
rollback function 348, 350, 382, 1634
room function 145, 146, 149, 458, 695
room-values function 458, 1838
Rotate Active Finders Editor
command 792
rotate-byte function 1172
rotate-ring function 995
round-to-single-precision
function 1172
rules
grammar 313
:rules keyword 308
run time parameters 454
run-shell-command function 1839
S
:s inspector command 39
safe-format-to-limited-
string function 996
safe-format-to-string
function 996
safe-locale-file-encoding
function 1844
safe-prinl-to-string
function 996
safe-princ-to-string
function 996
safety 105
save-argument-real-p
function 997
save-code-coverage-data
function 889
save-current-code-coverage
function 889
save-current-profiler-tree
function 174, 998
save-current-session
function 1000
save-image function 179, 191, 454, 460,
464, 1001
save-image-with-bundle
function 1010
save-tags-database function 821
save-universal-from-script
function 1011
saving images 179
sbchar accessor 440
sbchar function 1173
schar accessor 440
schedule-timer function 293, 1489
schedule-timer-milliseconds
function 1491
schedule-timer-relative
function 1493
schedule-timer-relative-mil-
seconds function 1494
scheduling of processes 295
security-description-string-
for-open-named-pipe
function 1910
segmentation violation in compiled
code 108
select function 346, 357, 1634
select SQL SQL operator 363
**Index**

Self-contained examples
- Asynchronous I/O 482
- COMM package 481
- DDE 482
- miscellaneous examples 483
- parser generator 482
- `save-image` in a Mac OS X application bundle 482
- socket streams 481
- SSL 481
- streams 482
- TCP sockets 481

**Semaphore** system class 1496
- `semaphore-acquire` function 288, 1496
- `semaphore-count` function 288, 1498
- `semaphore-name` function 288, 1498
- `semaphore-release` function 288, 1499
- `semaphore-wait-count` function 288, 1500

**send-message-to-java-host** function 1307

**sequencep** function 1174

**serial-port** function 1520

**serial-port-input-available-p** function 1522

**server-terminate** function 605

Set Default Remote Debugging
- Editor command 752
- `set-application-themed` function 1914
- `set-approaching-memory-limit-callback` function 1845
- `set-array-single-thread-p` function 1012
- `set-array-weak` function 165, 1012
- `set-automatic-gc-callback` function 150, 1846
- `set-blocking-gen-num` function 150, 1848
- `set-clos-initarg-checking` function 501
- `set-code-coverage-snapshot` function 940
- `setCurrentActivity` Java method 1356
- `set-debugger-options` function 773
- `set-default-character-element-type` function 439, 442, 443, 1174
- `set-default-generation` function 137, 146, 1014
- `set-default-remote-debugging-connection` function 34, 774
- `set-default-segment-size` function 151, 1851
- `set-delay-promotion` function 151, 1852
- `setErrorReporter` Java method 1347
- `set-expected-allocation-in-generation-2-after-gc` function 1853
- `setf cdr-assoc` function 1730
- `setf timer-name` function 1503
- `set-file-dates` function 1857
- `set-funcall-async-limit` function 1500
- `set-gc-parameters` function 139, 146, 1015
- `set-generation-2-gc-options` function 1858
- `set-gen-num-gc-threshold` function 150, 1860
- `setGuiErrorReporter` Java method 1347
- `set-hash-table-weak` function 165, 1017
- `set-java-field` function 1302
- `set-make-instance-argument-checking` function 502
- `set-maximum-memory` function 146, 1862
- `set-maximum-segment-size` function 148, 150, 1863
- `set-memory-check` function 1865
- `set-memory-exhausted-call-back` function 1866
- `setMessageHandler` Java method 1354
- `set-minimum-free-space` function 139, 146, 1019
- `set-prepared-statement-variables` function 1639
- `set-primitive-array-region` function 1248
- `set-process-profiling` function 169, 170, 1020
- `set-profiler-threshold` function 1022
- `set-promote-generation-1`
Index

function 1868
set-promotion-count function 1023
set-quit-when-no-windows function 1176
set-registry-value function 468, 1937
set-remote-debugging-connection function 33, 775
set-reserved-memory-policy function 1869
set-serial-port-state function 1523
set-signal-handler function 1870
set-spare-keeping-policy function 151, 1871
set-split-promotion function 1872
set-ssl-ctx-dh function 426, 608
set-ssl-ctx-options function 426
set-ssl-ctx-options function 610
set-ssl-ctx-password-callback function 1086
set-ssl-library-path function 420, 611
set-static-segment-size function 1874
sets-who function 1030
set-system-message-log function 1024
set-temp-directory function 468, 1875
setTextView Java method 1354
setup-atomic-funcall function 1876
setup-deliver-dynamic-library-for-java function 217, 1308
setup-field-accessor function 1312
setup-java-caller function 1313
setup-java-constructor function 1313
setup-java-interface-callbacks function 1253
setup-lisp-proxy function 1315
setup-profile function 168, 1025
set-verification-mode function 606
*sg-default-size* variable 1877
:sh inspector command 39
shared libraries 191, 459
shared library 191
shared object file 191
Shift JIS 444
short-float type 480, 700
short-namestring function 454
short-site-name function 453, 701
Show Paths From Editor command 116, 863
Show Paths To Editor command 1060
showBugFormLogs Java method 1351
shutdown 99
sid-string-to-user-name function 1916
simple-augmented-string type 1722
simple-augmented-string-p function 1722
simple-base-string type 439, 440, 641
simple-base-string-p function 1086
simple-bmp-string type 439, 1088
simple-bmp-string-p function 1089
simple-char type 1176
simple-char-p function 1177
simple-do-query macro 351, 381, 1640
SimpleInitLispWorks C function 1990
simple-int32-vector type 473, 1878
simple-int32-vector-length function 1879
simple-int32-vector-p function 1879
simple-int64-vector type 474, 1880
simple-int64-vector-length function 1880
simple-int64-vector-p function 1881
simple-lock-and-condition-variable-wait function 1501
simple-string type 439, 440, 702
simple-text-string type 439, 440, 1185
simple-text-string-p
Index

function 1186
single-float type 480, 703
single-form-form-parser function 822
single-form-with-options-form-parser function 822
single-threaded
arrays 276
hash tables 276
:size initarg 1884
:sjis external format 444
SLIME 3
slot-boundp-using-class generic function 249, 502
slot-makunbound-using-class generic function 249, 503
slot-value
atomic operations 299
:socket initarg 612
socket-error class 612
socket-stream class 418, 422, 612
socket-stream-address function 617
socket-stream-ctx function 431, 618
socket-stream-handshake function 618
socket-stream-peer-address function 619
socket-stream-shutdown function 620
socket-stream-socket accessor 613
socket-stream-ssl function 431, 621
socket-stream-ssl-side function 431, 622
software-type function 453, 703
software-version function 453, 704
sort-inspector-p generic function 1138
source level debugging 88, 116, 1040
source-debugging-on-p function 1031
*source-found-action* variable 185
:source-only keyword 308
space 105
special actions 164
special forms declare 104, 109, 651
specific-valid-file-encoding function 1882
*specific-valid-file-encodings* variable 1881
speed 105
splash screen 1898
split-sequence function 1177
split-sequence-if function 1178
split-sequence-if-not function 1179
SQL
database functions 365
database operators 365
direct specification 352
mode 375
stored procedure 352, 1567
sql function 368, 1642
SQL pseudo operators
 sql-boolean-operator 366, 368, 1660
 sql-function 365, 368, 1660
 sql-operator 366, 368, 1660
sql-boolean-operator SQL pseudo operator 366, 368, 1660
sql-connection-error condition 1643
sql-connection-error error 373
sql-database-data-error accessor 1644
sql-database-data-error condition 1643
sql-database-data-error error 372
sql-database-error condition 1643
sql-database-error error 372
*sql-enlarge-static* variable 1644
sql-error-database-message accessor 1644
sql-error-secondary-error-id accessor 1644
sql-expression function 368, 1645
sql-expression-object system class 1647
sql-fatal-error condition 1648
sql-fatal-error error 373
sql-function SQL pseudo operator 365, 368, 1660
sqlite-blob system class 395, 1648
sqlite-blob-length function 1650
sqlite-blob-p function 1650

2049
sqlite-close-blob function 395, 1650
sqlite-last-insert-rowid function 1649
sqlite-open-blob function 395, 1650
sqlite-raw-blob function 394, 1654
sqlite-raw-blob-length function 395, 1655
sqlite-raw-blob-p function 1655
sqlite-raw-blob-ref function 395, 1655
sqlite-raw-blob-valid-p function 395, 1655
sqlite-reopen-blob function 1650
*sql-libraries* variable 337, 1659
*sql-loading-verbose* variable 337, 349, 1660
sql-operation function 368, 1660
sql-operator function 368, 1662
sql-operator SQL pseudo operator 366, 368, 1660
sql-recording-p function 372, 1663
sql-stream function 372, 1664
sql-temporary-error condition 1665
sql-temporary-error error 372
sql-timeout-error condition 1665
sql-timeout-error error 373
sql-user-error condition 395, 396, 1653, 1659, 1665
sql-user-error error 372
square bracket syntax 359
ssl-add-client-ca function 427
ssl-cipher-get-bits function 427
ssl-cipher-get-name function 427
ssl-cipher-get-version function 427
ssl-cipher-pointer FLI type 427, 622
ssl-clear-num-renegotiations function 427
ssl-closed class 431, 623
ssl-condition class 431, 623
:ssl-configure-callback initarg 515, 565, 592, 613
ssl-ctrl function 428
:ssl-ctx initarg 613
ssl-ctx-add-client-ca function 428
ssl-ctx-add-extra-chain-cert function 428
ssl-ctx-ctrl function 428
ssl-ctx-get-max-cert-list function 428
ssl-ctx-get-mode function 428
ssl-ctx-get-options function 428
ssl-ctx-get-read-ahead function 428
ssl-ctx-get-verify-mode function 428
ssl-ctx-load-verify-locations function 428
ssl-ctx-need-temp-rsa function 428
ssl-ctx-pointer FLI type 427, 624
ssl-ctx-sess-get-cache-mode function 428
ssl-ctx-sess-get-cache-size function 428
ssl-ctx-sess-set-cache-size function 428
ssl-ctx-sess-set-cache-size function 428
ssl-ctx-set-client-ca-list function 428
ssl-ctx-set-max-cert-list function 428
ssl-ctx-set-mode function 428
ssl-ctx-set-options function 428
ssl-ctx-set-read-ahead function 428
ssl-ctx-set-tmp-dh function 428
ssl-ctx-set-temp-rsa function 428
ssl-ctx-use-certificate-file function 429
ssl-ctx-use-playskey-file function 429
ssl-ctx-use-rasp-ekey-file function 429
ssl-error class 431, 624
:ssl-error-callback initarg 515
ssl-failure class 431, 624
ssl-get-current-cipher function 429
ssl-get-max-cert-list function 429
ssl-get-mode function 429
ssl-get-options function 429
ssl-get-verify-mode function 429
Index

accessor 1885
storage-exhausted-type accessor 1885
str FLI type 1916 :
istream initarg 612
stream-advance-to-column generic function 1689
stream-check-eof-no-hang generic function 1690
stream-clear-input generic function 402, 1690
stream-clear-output generic function 403, 1691
stream-element-type generic function 400, 709
stream-file-position generic function 1692
stream-fill-buffer generic function 1692
stream-finish-output generic function 403, 1693
stream-flush-buffer generic function 1694
stream-force-output generic function 403, 1695
stream-fresh-line generic function 1695
stream-line-column generic function 403, 1696
stream-listen generic function 402, 1697
stream-output-width generic function 1698
stream-peek-char generic function 1698
stream-read-buffer generic function 1699
stream-read-byte generic function 1700
stream-read-char generic function 401, 1701
stream-read-char-no-hang generic function 1701
stream-read-line generic function 1702
stream-read-sequence generic function 1703
stream-read-timeout accessor 613
stream-read-timeout generic function 1704
streams defining new 400
directionality 400
example 400–405
input 401
instantiating 404
output 402
user defined 399
stream-start-line-p generic function 403, 1704
stream-terpri generic function 1705
stream-unread-char generic function 401, 1706
stream-write-buffer generic function 1706
stream-write-byte generic function 1707
stream-write-char generic function 402, 1708
stream-write-sequence generic function 1708
stream-write-string generic function 1709
stream-write-timeout accessor 613
string construction 441
string type 438, 439, 710
string types 438
string-append function 1182
string-append* function 1183
string-ip-address function 633
string-needs-n-prefix function 1668
string-prefix-with-n-if-needed function 1669
string-trim-whitespace function 1035
structure types async-io-state 412, 523
structurep function 1184
subtract-code-coverage-data function 839
superclass invalid 253
sweep 138
sweep-all-objects function 164, 1037
sweep-gen-num-objects function 1885
switch-open-tcp-stream-with-ssl-to-java function 634
switch-static-allocation function 136, 137, 1038
symbol macros
Index

+int32-0+ 1774
+int32-1+ 1774
+int64-0+ 1785
+int64-1+ 1785
*symbol-alloc-gen-num*
  variable 146, 164, 1039
symbol-dynamically-bound-p
  function 1039
symbolic query syntax 358
symbolic syntax in Common SQL 358
symbols
  allocation of 164
symeval-in-process function 295, 1503
Synchronization barriers 287
syntax, in Common SQL 358
system
  compile 1092
  defining 306–311
  introduction to 305–311
  load 1153
  members of 307
  plan 308
  print 1147
  rules 308–310
system classes
  barrier 1362
  client-remote-debugging 765
  condition-variable 1376
  ide-remote-debugging 765
  lock 1401
mailbox 1411
remote-debugging-connection 765
semaphore 1496
sql-expression-object 1647
sqlite-blob 395, 1648
sqlite-raw-blob 394, 1654
system commands
  running directly 1725
  running via a shell 1725
T
table-exists-p function 1670
tail call 113
tail call merging 113
tail call optimization 113
tail merge 113
tail recursion 113
tail-call 113
TCP/IP socket
  client side 408
  server side 407
SSL interface 418
teletype inspector 37
temp files 467
temporary files 467
*terminal-debugger-block-multiprocessing*
  variable 783
text-string type 438, 1185
text-string-p function 1186
threads 261
  allocation of 164
throw-an-exception function 1317
throw-if-tag-found macro 1040
time macro 711
timer-expired-p function 1504
timer-name function 1505
timers 293
  I/O 293
  input and output 293
  multiprocessing 293
  process 293
  threading issues 293
:tlsext-host-name initarg 613
toggle-source-debugging function 117, 1040
*to-java-host-stream*
  variable 1318
*to-java-host-stream-no-scroll* variable 1319
tools
  inspector 37
:top debugger command 21
top-level loop 5
total-allocation function 145, 1042
trace
  excessive output 60
  main chapter 47
  missing output 60
  not working 60
  troubleshooting 60
trace macro 713
traceable-dspec-p function 823
traced functions
  arguments 50
*traced-arglist*
  variable 50, 60, 1043
*traced-results*
  variable 50, 60, 714, 1043
*trace-indent-width*
  variable 59, 1044
*trace-level*
  variable 60, 1045

2053
trace-new-instances-on-access function 505
trace-on-access function 506
:trace-output keyword 54
*trace-output* variable 60, 705
*trace-print-circle* variable 60, 1046
*trace-print-length* variable 38, 60, 1047
*trace-print-level* variable 38, 60, 1048
*trace-print-pretty* variable 60, 1049
tracer
:after option 50
:allocation option 55
:before option 49
:break option 51
:break-on-exit option 51
commands available 49–56
definition specs 58
directing output 54
everting the stepper 52
:entrycond option 52
:eval-after option 51
:eval-before option 50
evaluating forms 49–51
example of use 47
:exitcond option 52
functions, tracing inside 55
information displayed 48
:inside option 55
invoking the debugger 51
memory allocation 55
methods, tracing 57
:process option 54
restricting to a process 54
:step option 52
traced function, arguments for 50
traced functions, results for 50
:trace-output option 54
*trace-verbose* variable 1050
tracing functions
inside other functions 55
tracing-enabled-p function 824
tracing-state function 825
transaction handling
in Common SQL 339, 350, 377
true function 1186
truename function 720
try-compact-in-generation function 144, 146, 1051
try-move-in-generation function 144, 146, 1052
tstr FLI type 1917
tty 1002
:type initarg 1885
typed-aref function 1886
types
16-bit-string 1082
8-bit-string 1081
accepting-handle 517
augmented-string 1722
base-char 436, 1084
base-character 1084
base-string 438, 641
bmp-char 436, 1086
bmp-string 438, 1088
character 436
code-coverage-data 870
code-coverage-file-stats 879
double-float 669
extended-char 1131
extended-character 1131
fixnum 477
int32 472, 1772
int64 474, 1783
java-vm-poi 1280
jboolean 1280
jbyte 1280
jchar 1280
jdouble 1280
jfloat 1280
jint 1280
jlong 1280
jni-env-poi 1281
jobject 1281
jshort 1280
long-float 678
mt-random-state 1158
prepared-statement 1629
short-float 480, 700
simple-augmented-string 1722
simple-base-string 439, 440, 641
simple-bmp-string 439, 1088
simple-char 1176
simple-int32-vector 473, 1878
simple-int64-vector 474, 1880
simple-string 439, 440, 702
simple-text-string 439, 440, 1185
single-float 480, 703
string 438, 439, 710
text-string 438, 1185
unlocked-queue 1055

U
:u inspector command 39
:ud inspector command 39, 40
unbreak-new-instances-on-access function 509
unbreak-on-access function 509
undefined-action macro 96, 1187
undefined-action-list macro 96, 1188
undefined-declaration function 1054
Unicode 435
:unicode external format 444
unicode-alpha-char-p function 451, 1188
unicode-alphanumericp function 451, 1189
unicode-both-case-p function 451, 1190
unicode-char-equal function 451, 1191
unicode-char-greaterp function 451, 1192
unicode-char-lesserp function 451, 1192
unicode-char-not-equal function 451, 1191
unicode-char-not-greaterp function 451, 1193
unicode-char-not-lesserp function 451, 1193
unicode-lower-case-p function 451, 1194
unicode-string-equal function 451, 1196
unicode-string-greaterp function 451, 1196
unicode-string-lesserp function 451, 1196
unicode-string-not-equal function 451, 1194
unicode-string-not-greaterp function 451, 1197
unicode-string-not-lesserp function 451, 1197
unicode-upper-case-p function 451, 1198
universal binaries
  helper functions 862, 997
  saving
advanced 897
  simply 1011
UNIX command
call-system 1724
call-system-showing-output 1727
open-pipe 1823
run-shell-command 1841
Unix commands
calling from Lisp 462
Unix functions
calling from Lisp 462
unixODBC 390
unlocked-queue type 1055
unlocked-queue-count function 964
unlocked-queue-peek function 964
unlocked-queue-read function 964
unlocked-queue-ready function 964
unlocked-queue-send function 964
unlocked-queue-size function 964
unnotice-fd function 1506
*unresolved-messages* 1080
unschedule-timer function 1506
untrace macro 721
untrace-new-instances-on-access function 510
untrace-on-access function 510
unwind-protect-blocking-interrupts macro 289, 1055
unwind-protect-blocking-interrupts-in-cleanups macro 289, 1057
update-instance-for-different-class function 722
update-instance-for-redefined-class function 723
update-instance-from-records generic function 357, 1671
update-objects-joins function 1672
update-record-from-slot generic function 357, 1675
update-records function 348, 349, 381, 1673
update-records-from-instance generic function 357, 1674
update-slot-from-record generic function 357, 1675
URL
  opening 1827
:use listener command 7
*use-n-syntax-for-non-ascii-strings* variable 1676
user defined stream 399
user-homedir-pathname function 227, 466, 1764
user-name-to-sid-string function 1919
user-preferece function 464, 1199
UTF-16 external format 444
:utf-16be external format 446, 830
:utf-16le external format 446, 830
:utf-16-native external format 446, 830
:utf-16-reversed external format 446, 830
:utf-32 external format 444
:utf-8 external format 444
utilities in Common SQL 370

V
:v debugger command 16
validate-superclass generic function 251, 252
valid-external-format-p function 838
variable-information function 1058
variables
$(dollar) 38
$$ 38
$$$ 38
*active-finders* 791
*autoload-asdf-integration* 1083
*background-input* 858
*background-output* 858
*background-query-io* 858
*binary-file-type* 649, 1723
*binary-file-types* 649, 1723
*browser-location* 1089
*cache-table-queries-default* 1532
*check-network-server* 1731
*compiler-break-on-error* 887
*connect-if-exists* 1541
*current-process* 262, 1380
*debug-initialization-errors-in-snap-shot* 1736
*debug-io* 23, 705
*debug-print-length* 23, 738
*debug-print-level* 24, 738
*default-action-list-sort-time* 97, 1103
*default-client-remote-debugging-server-port* 27, 739
*default-database* 1549
*default-database-type* 1550
*default-ide-remote-debugging-server-port* 27, 740
*default-libraries* 1079
*default-package-use-list* 901
*default-process-priority* 1389
*default-profiler-collapse* 901
*default-profiler-cutoff* 902
*default-profiler-limit* 902
*default-profiler-sort* 903
*default-stack-group-list-length* 164, 1738
*default-update-objects-max-len* 1550
*defstruct-generates-print-object-method* 189
*defsystem-verbose* 1115
*describe-length* 39, 1117
*describe-level* 38, 1117
*describe-print-length* 38, 1119
*describe-print-level* 38, 1119
*directory-link-transparency* 664, 1745
*disable-trace* 914
*dspec-classes* 804
*enter-debugger-directly* 1124
*error-output* 705
*extended-spaces* 1203, 1748
*external-formats* 1133
*features* 670
*file-encoding-detection-algorithm* 448, 1749
*file-eol-style-detection-algorithm* 448, 1751
*filename-pattern-encoding-matches* 1752
Index

*traced-results* 50, 60, 714, 1043
*trace-indent-width* 59, 1044
*trace-level* 60, 1045
*trace-output* 60, 705
*trace-print-circle* 60, 1046
*trace-print-length* 38, 60, 1047
*trace-print-level* 38, 60, 1048
*trace-print-pretty* 60, 1049
*use-n-syntax-for-non-ascii-strings* 1676
vector-pop function 266, 276
vector-push function 266, 276
vector-push-extend function 266, 276
verbose backtrace 14
verify-java-caller function 1320
verify-java-callers function 1321
verify-lisp-proxies function 1324
verify-lisp-proxy function 1324
virtual (ordinary) slots 355
virtual time 169
Visit Tags File Editor command 792

W
wait-for-connection generic function 1919
waitForInitialization method 1336
wait-for-input-streams function 1888
wait-for-input-streams-returning-first function 1889
wait-for-wait-state-collection function 411, 635
wait-processing-events function 1508
wait-serial-port-state function 1524
wait-state-collection class 411, 636
wait-state-collection-stop-loop function 411, 415, 636
weak arrays 1012
hash tables 682
weak hash tables 682
weak pointers 1012
web browser 1827
:when keyword 55
when-let macro 1201
when-let* macro 1202
whitespace-char-p function 1203
who-binds function 1059
who-calls function 1060
who-references function 1061
who-sets function 1061
Windows code page 936 444
Windows event log 1909
Windows registry API 468, 1923
Windows themes 1914
windows-cp936 444
:windows-cp936 external format 444
with-action-item-error-handling macro 1204
with-action-list-mapping macro 1205
with-code-coverage-generation macro 1062
with-dde-conversation macro 323, 1968
with-debugger-wrapper macro 23, 785
with-ensuring-gethash macro 1063
with-exclusive-lock macro 277, 1509
with-hash-table-locked macro 266, 267, 1065
with-heavy-allocation macro 146, 1066
with-interrupts-blocked macro 289, 1510
with-lock macro 277, 1510
with-modification-change macro 1890
with-modification-check-macro macro 1891
with-noticed-socket-stream macro 637
with-other-threads-disabled macro 289, 1892
without-code-coverage macro 1071
without-interrupts macro 289, 290, 1512
without-preemption macro 289,
Index

with-output-to-fasl-file  
macro 1067

with-output-to-string  
macro 723

with-registry-key  
macro 468, 1938

with-remote-debugging-connection  
function 33

with-remote-debugging-connection  
macro 788

with-remote-debugging-spec  
macro 789

with-ring-locked  
macro 1071

with-sharing-lock  
macro 277, 1511

with-sqlite-blob  
macro 395, 1677

with-stream-input-buffer  
macro 1710

with-stream-output-buffer  
macro 1712

with-transaction  
macro 348, 350, 1678

with-unique-names  
macro 1206

with-windows-event-log-event-source  
macro 1921

write-java-class-definitions-to-file  
function 1326

write-java-class-definitions-to-stream  
function 1326

write-serial-port-char  
function 1524

write-serial-port-string  
function 1525

write-string-with-properties  
function 1072

:write-timeout  
initarg 613

wstr  
FLI type 1918

X
xrefs 860, 863, 1040, 1059, 1060

Y
yellow pages 578

yield  
function 1514
Index