Contents

Preface  xlvii

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

4  The REPL Inspector  25
   Describe  25
Contents

Inspect 26
Inspection modes 28

5 The Trace Facility 35
Simple tracing 35
Tracing options 37
Example 44
Tracing methods 45
Trace variables 46
Troubleshooting tracing 47

6 The Advice Facility 49
Defining advice 50
Combining the advice 51
Removing advice 52
Advice for macros and methods 53
Examples 55
Advice functions and macros 58

7 Dspecs: Tools for Handling Definitions 59
Dspecs 59
Forms of dspecs 60
Dspec namespaces 61
Types of relations between definitions 65
Details of system dspec classes and aliases 66
Subfunction dspecs 68
Tracking definitions 69
Finding locations 71
Users of location information 71

8 Action Lists 77
Defining action lists and actions 78
Exception handling variables 78
Other variables 79
Diagnostic utilities 79
Examples 79
Standard Action Lists 81
9 The Compiler 83
  Compiling a function 84
  Compiling a source file 85
  Compiling a form 85
  How the compiler works 85
  Compiler control 86
  Declare, proclaim, and declaim 91
  Optimizing your code 92
  Compiler parameters affecting LispWorks 98

10 Code Coverage 101
  Using Code Coverage 101
  Manipulating code coverage data 103
  Preventing code generation for some forms 104
  Code coverage and multithreading 105
  Counting overflow 105
  Memory usage and code speed 105
  Understanding the code coverage output 106
  Coloring code that has changed 109

11 Storage Management 111
  Introduction 111
  Guidance for control of the storage management system 112
  Memory Management in 32-bit LispWorks 117
  Memory Management in 64-bit LispWorks 128
  Common Memory Management Features 133

12 The Profiler 137
  What the profiler does 137
  Setting up the profiler 138
  Running the profiler 139
  Profiler output 141
  Interpretation of profiling results 142
  Profiling pitfalls 142
  Profiling and garbage collection 143

13 Customization of LispWorks 145
Contents

Introduction 145
Configuration and initialization files 146
Saving a LispWorks image 147
Saved sessions 150
Load and open your files on startup 152
Customizing the editor 153
Finding source code 154
Controlling redefinition warnings 155
Specifying the initial working directory 155
Using ! for :redo 156
Customizing LispWorks for use with your own code 156
Structure printing 157
Configuring the printer 157

14 LispWorks as a dynamic library 159
Introduction 159
Creating a dynamic library 160
Initialization of the dynamic library 161
Relocation 162
Multiprocessing in a dynamic library 163
Unloading a dynamic library 163

15 Java interface 165
Types and conversion between Lisp and Java 166
Calling from Lisp to Java 168
Calling from Java to Lisp 173
Working with Java arrays 178
Initialization of the Java interface 180
Utilities and administration 181
Loading a LispWorks dynamic library into Java 184
CLOS partial integration 185
Java interface performance issues 186

16 Android interface 189
Delivering for Android 190
Directories on Android 193
The Othello demo for Android 193
17 iOS interface 207
   Delivering for iOS 207
   Initializing LispWorks 208
   Using Objective-C from Lisp 209
   The Othello demo for iOS 209

18 The Metaobject Protocol 213
   Metaobject features incompatible with AMOP 213
   Metaobject features additional to AMOP 216
   Common problems when using the MOP 216
   Implementation of virtual slots 217

19 Multiprocessing 225
   Introduction to processes 225
   Processes basics 226
   Atomicity and thread-safety of the LispWorks implementation 229
   Locks 232
   Modifying a hash table with multiprocessing 235
   Process Waiting and communication between processes 235
   Synchronization between threads 240
   Killing a process, interrupts and blocking interrupts 244
   Timers 248
   Process properties 249
   Other processes functions 250
   Native threads and foreign code 252
   Low level operations 253

20 Common Defsystem and ASDF 259
   Introduction 259
   Defining a system 260
   Using ASDF 265

21 The Parser Generator 267
   Introduction 267
   Grammar rules 267
   Functions defined by defparser 269
   Error handling 270
Contents

Interface to lexical analyzer 270
Example 271

22 Dynamic Data Exchange 275
Introduction 275
Client interface 277
Server interface 281

23 Common SQL 285
Introduction 285
Initialization 288
Functional interface 297
Object oriented interface 305
Symbolic SQL syntax 310
Working with date fields 317
SQL I/O recording 319
Error handling in Common SQL 320
Using MySQL 321
Using Oracle 326
Oracle LOB interface 327
Using ODBC 337

24 User Defined Streams 339
Introduction 339
An illustrative example of user defined streams 339

25 TCP and UDP socket communication and SSL 347
Running a server that accepts connections 347
Connecting to a server 348
Specifying the target for connecting and binding a socket 348
Information about IP addresses 349
Waiting on a socket stream 350
Special considerations 350
Socket Stream SSL interface 351
Asynchronous I/O 361
Socket streams with Java sockets and SSL on Android 369
26  **Internationalization: characters, strings and encodings**  373

   Introduction  373  
   Unicode support  373  
   Character and String types  374  
   String accessors  378  
   String Construction  379  
   External Formats  381  
   16-bit External formats guide  383  
   External Formats and File Streams  385  
   External Formats and the Foreign Language Interface  388  
   Unicode character and string functions  388

27  **LispWorks' Operating Environment**  391

   The Operating System  391  
   Site Name  391  
   The Lisp Image  392  
   The Command Line  392  
   Address Space and Image Size  396  
   Startup relocation  396  
   Calling external programs  400  
   Snapshot debugging of startup errors  401  
   System message log  401  
   Exit status  401  
   Creating a new executable with code preloaded  401  
   User Preferences  402  
   File system interface  403  
   Special locations in the file system  404  
   Accessing the Windows registry  406  
   Pathname comparison on Mac OS X  407

28  **Miscellaneous Utilities**  409

   Object addresses and memory  409  
   Optimized integer arithmetic and integer vector access  410  
   Transferring large amounts of data  414  
   Rings  414  
   Conditional throw and checking for catch in the dynamic environment  414
Contents

Checking for a dynamic binding 414

29 64-bit LispWorks 415

Introduction 415
Heap size 416
Architectural constants 417
Speed 417
Memory Management and cl:room 417
Greater allocation expected in 64-bit LispWorks 417
Float types 418
External libraries 418

30 Self-contained examples 419

COMM examples 419
Streams examples 420
DDE examples 420
Parser generator examples 420
Examples for save-image in a Mac OS X application bundle 420
Miscellaneous examples 421

31 The CLOS Package 423

break-new-instances-on-access 423
break-on-access 424
class-extra-initargs 425
compute-class-potential-initargs 426
compute-discriminating-function 428
compute-effective-method-function-from-classes 428
copy-standard-object 431
funcallable-standard-object 432
process-a-class-option 433
process-a-slot-option 435
replace-standard-object 437
set-clos-initarg-checking 439
set-make-instance-argument-checking 440
slot-boundp-using-class 440
slot-makunbound-using-class 441
slot-value-using-class 442
trace-new-instances-on-access 443
trace-on-access 444
unbreak-new-instances-on-access 447
unbreak-on-access 447
untrace-new-instances-on-access 448
untrace-on-access 448

32 The COMM Package 451
accept-tcp-connections-creating-async-io-states 451
accepting-handle 454
accepting-handle-collection 455
accepting-handle-name 456
accepting-handle-local-port 456
accepting-handle-socket 457
accepting-handle-user-info 458
apply-in-wait-state-collection-process 458
async-io-state 459
async-io-state-abort 460
async-io-state-abort-and-close 462
async-io-state-address 463
async-io-state-buffered-data-length 464
async-io-state-discard 464
async-io-state-finish 465
async-io-state-get-buffered-data 466
async-io-state-peer-address 467
async-io-state-read-buffer 468
async-io-state-read-with-checking 470
async-io-state-receive-message 473
async-io-state-send-message 475
async-io-state-send-message-to-address 477
async-io-state-write-buffer 479
attach-ssl 480
call-wait-state-collection 482
close-accepting-handle 482
close-async-io-state 483
close-wait-state-collection 484
create-and-run-wait-state-collection 484
create-async-io-state 487
create-async-io-state-and-connected-tcp-socket 489
create-async-io-state-and-connected-udp-socket 491
create-async-io-state-and-udp-socket 493
connect-to-tcp-server 495
destroy-ssl 498
destroy-ssl-ctx 498
detach-ssl 498
do-rand-seed 500
ensure-ssl 500
get-default-local-ipv6-address 501
get-host-entry 502
get-ip-default-zone-id 505
get-service-entry 505
get-socket-address 507
get-socket-peer-address 508
get-verification-mode 508
ip-address-string 509
ipv6-address 510
ipv6-address-p 511
ipv6-address-scope-id 511
ipv6-address-string 512
make-ssl-ctx 512
loop-processing-wait-state-collection 513
make-wait-state-collection 514
open-tcp-stream 514
open-tcp-stream-using-java 518
openssl-version 522
parse-ipv6-address 523
pem-read 524
read-dhparams 526
replace-socket-stream-socket 527
server-terminate 528
set-verification-mode 529
set-ssl-ctx-dh 531
set-ssl-ctx-options 532
set-ssl-ctx-password-callback 533
set-ssl-library-path 534
socket-error 535
socket-stream 535
socket-stream-address 541
socket-stream-ctx 541
socket-stream-peer-address 542
Contents

socket-stream-shutdown 543
socket-stream-ssl 544
ssl-cipher-pointer 544
ssl-cipher-pointer-stack 544
ssl-closed 545
ssl-condition 545
ssl-ctx-pointer 546
ssl-error 546
ssl-failure 546
ssl-new 547
ssl-pointer 547
ssl-x509-lookup 548
start-up-server 548
start-up-server-and-mp 554
string-ip-address 554
switch-open-tcp-stream-with-ssl-to-java 555
wait-for-wait-state-collection 556
wait-state-collection 557
wait-state-collection-stop-loop 557
with-noticed-socket-stream 558

33 The COMMON-LISP Package 561

apropos 561
apropos-list 562
base-string 563
simple-base-string 563
close 563
c coerce 564
compile 565
compile-file 567
concatenate 572
declaim 572
declare 573
defclass 578
defpackage 582
describe 584
directory 584
disassemble 589
documentation 589
update-instance-for-redefined-class 641
with-output-to-string 641

34 The COMPILER Package 643
deftransform 643

35 The DBG Package 647
*debug-print-length* 647
*debug-print-level* 648
executable-log-file 649
*hidden-packages* 650
log-bug-form 652
logs-directory 653
output-backtrace 654
*print-binding-frames* 655
*print-catch-frames* 657
*print-handler-frames* 658
*print-invisible-frames* 660
*print-open-frames* 661
*print-restart-frames* 662
set-debugger-options 663
*terminal-debugger-block-multiprocessing* 664
with-debugger-wrapper 666

36 The DSPEC Package 671
*active-finders* 671
at-location 672
canonicalize-dspec 673
def 674
define-dspec-alias 675
define-dspec-class 676
define-form-parser 679
discard-source-info 683
dspec-class 683
*dspec-classes* 684
dspec-defined-p 684
dspec-definition-locations 685
dspec-equal 686
dspec-name 686
Contents

dspec-primary-name 687
 dspec-progenitor 688
 dspec-subclass-p 688
 dspec-undefiner 689
 find-dspec-locations 689
 find-name-locations 690
 get-form-parser 691
 local-dspec-p 692
 location 693
 name-defined-dspecs 693
 name-definition-locations 694
 name-only-form-parser 695
 parse-form-dspec 696
 record-definition 696
 *record-source-files* 698
 *redefinition-action* 698
 replacement-source-form 699
 save-tags-database 700
 single-form-form-parser 701
 single-form-with-options-form-parser 702
 traceable-dspec-p 703
 tracing-enabled-p 703
 tracing-state 704

37 The EXTERNAL-FORMAT Package 707

:bmp 707
:bmp-native 707
:bmp-reversed 707
:unicode 709
:utf-16 710
:utf-16be 710
:utf-16le 710
:utf-16-native 710
:utf-16-reversed 710
:utf-32 711
:utf-32le 711
:utf-32be 711
:utf-32-native 711
:utf-32-reversed 711
char-external-code 713
decode-external-string 714
encode-lisp-string 715
external-format-error 715
external-format-foreign-type 716
external-format-type 716
find-external-char 717
valid-external-format-p 718

38 The HCL Package 719
add-code-coverage-data 719
reverse-subtract-code-coverage-data 719
subtract-code-coverage-data 719
destructive-add-code-coverage-data 719
destructive-subtract-code-coverage-data 719
destructive-reverse-subtract-code-coverage-data 719
add-special-free-action 721
add-symbol-profiler 722
allocation-in-gen-num 723
analysing-special-variables-usage 724
android-funcall-in-main-thread 728
android-funcall-in-main-thread-list 728
android-get-current-activity 729
*android-main-process-for-testing* 730
android-main-thread-p 731
any-capi-window-displayed-p 732
array-single-thread-p 732
array-weak-p 733
augment-environment 733
avoid-gc 735
*background-input* 736
*background-output* 736
*background-query-io* 736
binds-who 738
block-promotion 739
building-universal-intermediate-p 740
calls-who 741
cd 742
change-directory 743
check-fragmentation 743
clean-down 744
clean-generation-0 746
clear-code-coverage 747
reset-code-coverage 747
restore-code-coverage-data 747
code-coverage-data 748
code-coverage-data-generate-coloring-html 750
code-coverage-data-generate-statistics 756
code-coverage-file-stats 757
code-coverage-set-editor-colors 760
code-coverage-set-editor-default-data 761
code-coverage-set-html-background-colors 762
collect-generation-2 763
collect-highest-generation 764
*compiler-break-on-error* 765
compile-file-if-needed 766
copy-code-coverage-data 767
copy-current-code-coverage 767
load-code-coverage-data 767
save-code-coverage-data 767
save-current-code-coverage 767
copy-to-weak-simple-vector 769
create-macos-application-bundle 770
create-temp-file 773
open-temp-file 773
create-universal-binary 775
current-stack-length 775
declaration-information 776
*default-package-use-list* 777
*default-profiler-collapse* 778
*default-profiler-cutoff* 778
*default-profiler-limit* 779
*default-profiler-sort* 779
defglobal-parameter 780
defglobal-variable 780
delete-advice 781
delivered-image-p 782
deliver-to-android-project 783
*disable-trace* 788
do-profiling 788
editor-color-code-coverage 791
enlarge-generation 794
enlarge-static 795
ensure-hash-entry 796
error-situation-forms 797
expand-generation-1 798
extend-current-stack 799
extended-time 800
fasl-error 802
fast-directory-files 803
fdf-handle-directory-p 803
fdf-handle-directory-string 803
fdf-handle-last-access 803
fdf-handle-last-modify 803
fdf-handle-link-p 803
fdf-handle-size 803
fdf-handle-writable-p 803
file-string 805
file-writable-p 806
filter-code-coverage-data 807
find-object-size 808
find-throw-tag 809
finish-heavy-allocation 810
flag-not-special-free-action 810
flag-special-free-action 811
function-information 812
get-code-coverage-delta 814
reset-code-coverage-snapshot 814
set-code-coverage-snapshot 814
gc-generation 815
gc-if-needed 818
generate-code-coverage 819
get-default-generation 822
get-gc-parameters 822
get-temp-directory 824
get-working-directory 824
gethash-ensuring 825
*handle-existing-defpackage* 827
*handle-old-in-package* 828
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>handle-old-in-package-used-as-make-package</em></td>
<td>829</td>
</tr>
<tr>
<td>hash-table-weak-kind</td>
<td>829</td>
</tr>
<tr>
<td>load-data-file</td>
<td>830</td>
</tr>
<tr>
<td><em>load-fasl-or-lisp-file</em></td>
<td>833</td>
</tr>
<tr>
<td>make-ring</td>
<td>834</td>
</tr>
<tr>
<td>make-unlocked-queue</td>
<td>836</td>
</tr>
<tr>
<td>unlocked-queue-read</td>
<td>836</td>
</tr>
<tr>
<td>unlocked-queue-ready</td>
<td>836</td>
</tr>
<tr>
<td>unlocked-queue-send</td>
<td>836</td>
</tr>
<tr>
<td>map-ring</td>
<td>837</td>
</tr>
<tr>
<td>mark-and-sweep</td>
<td>838</td>
</tr>
<tr>
<td><em>max-trace-indent</em></td>
<td>840</td>
</tr>
<tr>
<td>merge-code-coverage-data</td>
<td>841</td>
</tr>
<tr>
<td>destructive-merge-code-coverage-data</td>
<td>841</td>
</tr>
<tr>
<td>modify-hash</td>
<td>842</td>
</tr>
<tr>
<td>normal-gc</td>
<td>843</td>
</tr>
<tr>
<td><em>packages-for-warn-on-redefinition</em></td>
<td>844</td>
</tr>
<tr>
<td>parse-float</td>
<td>845</td>
</tr>
<tr>
<td>position-in-ring</td>
<td>846</td>
</tr>
<tr>
<td>position-in-ring-forward</td>
<td>846</td>
</tr>
<tr>
<td>print-profile-list</td>
<td>848</td>
</tr>
<tr>
<td><em>print-string</em></td>
<td>851</td>
</tr>
<tr>
<td>profile</td>
<td>851</td>
</tr>
<tr>
<td><em>profiler-threshold</em></td>
<td>853</td>
</tr>
<tr>
<td><em>profile-symbol-list</em></td>
<td>854</td>
</tr>
<tr>
<td>profiler-tree-from-function</td>
<td>854</td>
</tr>
<tr>
<td>profiler-tree-to-function</td>
<td>855</td>
</tr>
<tr>
<td>reduce-memory</td>
<td>856</td>
</tr>
<tr>
<td>references-who</td>
<td>858</td>
</tr>
<tr>
<td>remove-special-free-action</td>
<td>858</td>
</tr>
<tr>
<td>remove-symbol-profiler</td>
<td>859</td>
</tr>
<tr>
<td>reset-profiler</td>
<td>860</td>
</tr>
<tr>
<td>reset-ring</td>
<td>861</td>
</tr>
<tr>
<td>ringp</td>
<td>861</td>
</tr>
<tr>
<td>ring-length</td>
<td>861</td>
</tr>
<tr>
<td>ring-name</td>
<td>862</td>
</tr>
<tr>
<td>ring-pop</td>
<td>862</td>
</tr>
<tr>
<td>ring-push</td>
<td>863</td>
</tr>
<tr>
<td>ring-ref</td>
<td>864</td>
</tr>
<tr>
<td>rotate-ring</td>
<td>865</td>
</tr>
</tbody>
</table>
save-argument-real-p 866
save-current-session 867
save-image 868
save-image-with-bundle 877
save-universal-from-script 878
set-array-single-thread-p 879
set-array-weak 880
set-default-generation 881
set-gc-parameters 882
set-hash-table-weak 884
set-minimum-free-space 886
set-process-profiling 887
set-profiler-threshold 889
set-promotion-count 890
set-system-message-log 891
set-up-profiler 892
sets-who 896
source-debugging-on-p 896
start-gc-timing 897
stop-gc-timing 897
get-gc-timing 897
start-profiling 898
stop-profiling 900
sweep-all-objects 901
switch-static-allocation 902
*symbol-alloc-gen-num* 903
symbol-dynamically-bound-p 903
throw-if-tag-found 904
toggle-source-debugging 905
total-allocation 906
*traced-arglist* 907
*traced-results* 908
*trace-indent-width* 909
*trace-level* 910
*trace-print-circle* 911
*trace-print-length* 911
*trace-print-level* 912
*trace-print-pretty* 913
*trace-verbose* 914
try-compact-in-generation 915
Contents

try-move-in-generation 917
unlocked-queue 918
unwind-protect-blocking-interrupts 919
unwind-protect-blocking-interrupts-in-cleanups 920
variable-information 921
who-binds 923
who-calls 924
who-references 924
who-sets 925
with-code-coverage-generation 926
with-ensuring-gethash 926
with-hash-table-locked 928
with-heavy-allocation 929
with-output-to-fasl-file 930
dump-form 930
dump-forms-to-file 930
with-ring-locked 934
without-code-coverage 935

39 The LINK-LOAD Package 937
break-on-unresolved-functions 937
foreign-symbol-address 938
get-foreign-symbol 939
lisp-name-to-foreign-name 940
read-foreign-modules 941

40 The LISPWORKS Package 943
8-bit-string 943
16-bit-string 944
append-file 944
appendf 945
*autoload-asdf-integration* 945
base-character 946
base-character-p 946
base-char-p 947
base-char-code-limit 947
base-string-p 948
simple-base-string-p 948
bmp-char 948
Contents

bmp-char-p 949
bmp-string 950
simple-bmp-string 950
bmp-string-p 951
simple-bmp-string-p 951
*browser-location* 951
call-next-advice 952
choose-unicode-string-hash-function 954
compile-system 954
concatenate-system 956
copy-file 958
current-pathname 959
defadvice 960
*default-action-list-sort-time* 964
*default-character-element-type* 964
define-action 965
define-action-list 967
defsystem 969
*defsystem-verbose* 975
delete-directory 976
deliver 976
*describe-length* 977
*describe-level* 978
*describe-print-length* 980
*describe-print-level* 980
dll-quit 980
dotted-list-length 983
dotted-list-p 984
do-nothing 984
*enter-debugger-directly* 985
environment-variable 985
ererrno-value 987
eexample-file 987
example-compile-file 988
example-edit-file 989
example-load-binary-file 990
execute-actions 990
extended-char 992
extended-character 992
extended-character-p 992
Contents

extended-char-p 993
*external-formats* 993
false 995
file-directory-p 995
find-regexp-in-string 996
function-lambda-list 998
get-inspector-values 999
sort-inspector-p 999
get-unix-error 1002
*grep-command* 1002
*grep-command-format* 1003
*grep-fixed-args* 1004
*handle-existing-action-in-action-list* 1004
*handle-existing-action-list* 1005
*handle-missing-action-list* 1005
*handle-missing-action-in-action-list* 1006
*handle-warn-on-reredifinition* 1006
hardcopy-system 1007
*init-file-name* 1008
*inspect-through-gui* 1009
lisp-image-name 1009
*lispworks-directory* 1010
load-all-patches 1012
load-system 1013
make-unregistered-action-list 1014
make-mt-random-state 1016
mt-random 1017
*mt-random-state* 1017
mt-random-state 1018
mt-random-state-p 1018
pathname-location 1019
precompile-regexp 1020
print-actions 1020
print-action-lists 1021
*print-command* 1022
*print-nickname* 1022
*prompt* 1022
quit 1023
rebinding 1025
regexp-find-symbols 1026
Contents

remove-advice 1027
removef 1029
*require-verbose* 1029
round-to-single-precision 1030
sbchar 1031
sequencep 1031
set-default-character-element-type 1032
set-quit-when-no-windows 1033
simple-char 1034
simple-char-p 1034
split-sequence 1035
split-sequence-if 1036
split-sequence-if-not 1037
start-tty-listener 1038
stchar 1039
string-append 1039
string-append* 1040
structurep 1041
text-string 1042
simple-text-string 1042
text-string-p 1043
simple-text-string-p 1043
true 1043
undefine-action 1044
undefine-action-list 1045
unicode-alpha-char-p 1045
unicode-alphanumeric-p 1046
unicode-both-case-p 1047
unicode-char-equal 1048
unicode-char-not-equal 1048
unicode-char-greaterp 1049
unicode-char-lessp 1049
unicode-char-not-greaterp 1050
unicode-char-not-lessp 1050
unicode-lower-case-p 1051
unicode-string-equal 1051
unicode-string-greaterp 1053
unicode-string-lessp 1053
unicode-string-not-greaterp 1054
Contents

41 The LW-JI Package 1065

call-java-method 1065
call-java-method-error 1066
catching-java-exceptions 1067
catching-exceptions-bind 1067
check-lisp-calls-initialized 1069
create-instance-from-jobject 1069
create-instance-jobject-list 1070
create-instance-jobject 1070
create-java-object 1072
create-java-object-error 1073
default-constructor-arguments 1073
default-name-constructor 1074
define-field-accessor 1075
define-java-caller 1076
define-java-constructor 1076
define-java-callers 1079
define-lisp-proxy 1081
ensure-lisp-classes-from-tree 1089
ensure-supers-contain-java.lang.object 1091
field-access-exception 1092
field-exception 1093
find-java-class 1094
format-to-java-host 1095
generate-java-class-definitions 1095
get-java-virtual-machine 1101
get-jobject 1102
ensure-is-jobject 1102
get-primitive-array-region 1103

unicode-string-not-lessp 1054
unicode-upper-case-p 1055
user-preference 1056
when-let 1058
when-let* 1059
whitespace-char-p 1060
with-action-item-error-handling 1061
with-action-list-mapping 1062
with-unique-names 1063
Contents

set-primitive-array-region 1103
get-superclass-and-interfaces-tree 1105
import-java-class-definitions 1106
init-java-interface 1108
intern-and-export-list 1113
jaref 1114
java-array-element-type 1115
java-array-error 1116
java-array-indices-error 1117
java-array-length 1118
java-array-simple-error 1118
java-bad-jobject 1119
java-definition-error 1120
java-class-error 1120
java-method-error 1120
java-field-error 1120
java-exception 1121
java-field-setting-error 1122
java-id-exception 1123
java-instance-without-jobject-error 1123
java-interface-error 1124
java-low-level-exception 1125
java-method-exception 1125
java-normal-exception 1126
java-not-a-java-object-error 1127
java-not-an-array-error 1127
java-object-array-element-type 1128
java-objects-eq 1129
java-out-of-bounds-error 1130
java-storing-wrong-type-error 1130
java-primitive-array-element-type 1130
java-serious-exception 1131
java-type-to-lisp-array-type 1132
lisp-array-type-to-java-type 1132
java-vm-poi 1133
jboolean 1134
jbyte 1134
jchar 1134
jdouble 1134
jfloat 1134
Contents

jint 1134
jlong 1134
jshort 1134
jni-env-poi 1134
jobject 1135
jobject-class-name 1135
jobject-ensure-global 1136
jobject-of-class-p 1137
jobject-p 1138
jobject-pretty-class-name 1139
jobject-string 1139
jobject-to-lisp 1140
jvref 1141
lisp-java-instance-p 1143
lisp-to-jobject 1144
make-java-array 1144
make-java-instance 1145
make-lisp-proxy 1146
make-lisp-proxy-with-overrides 1146
map-java-object-array 1149
primitive-array-to-lisp-array 1153
lisp-array-to-primitive-array 1153
read-java-field 1156
checked-read-java-field 1156
set-java-field 1156
check-java-field 1156
java-field-class-name-for-setting 1156
record-java-class-lisp-symbol 1158
report-error-to-java-host 1159
reset-java-interface-for-new-jvm 1160
send-message-to-java-host 1161
setup-field-accessor 1161
setup-java-caller 1163
setup-java-constructor 1163
setup-lisp-proxy 1164
standard-java-object 1165
throw-an-exception 1167
verify-java-caller 1168
verify-java-callers 1169
verify-lisp-proxy 1172
verify-lisp-proxies 1172
write-java-class-definitions-to-file 1174
write-java-class-definitions-to-stream 1174

42  Java classes and methods  1179
com.lispworks.LispCalls 1179
com.lispworks.LispCalls.callIntV 1180
com.lispworks.LispCalls.callIntA 1180
com.lispworks.LispCalls.callDoubleV 1180
com.lispworks.LispCalls.callDoubleA 1180
com.lispworks.LispCalls.callObjectV 1180
com.lispworks.LispCalls.callObjectA 1180
com.lispworks.LispCalls.callVoidV 1180
com.lispworks.LispCalls.callVoidA 1180
com.lispworks.LispCalls.createLispProxy 1182
com.lispworks.LispCalls.checkLispSymbol 1183

43  Android Java classes and methods  1185
com.lispworks.Manager 1185
com.lispworks.Manager.init 1187
com.lispworks.Manager.status 1189
com.lispworks.Manager.init_result_code 1190
com.lispworks.Manager.mInitErrorString 1192
com.lispworks.Manager.loadLibrary 1192
com.lispworks.Manager.LispErrorReporter 1193
com.lispworks.Manager.setErrorReporter 1193
com.lispworks.Manager.LispGuiErrorReporter 1193
com.lispworks.Manager.setGuiErrorReporter 1193
com.lispworks.Manager.clearBugFormLogs 1196
com.lispworks.Manager.mMaxErrorLogsNumber 1196
com.lispworks.Manager.showBugFormLogs 1197
com.lispworks.Manager.addMessage 1198
com.lispworks.Manager.mMessagesMaxLength 1198
com.lispworks.Manager.setMessageHandler 1200
com.lispworks.Manager.MessageHandler 1200
com.lispworks.Manager.setTextView 1200
com.lispworks.Manager.getClassLoader 1201
com.lispworks.Manager.getApplicationContext 1201
com.lispworks.Manager.setCurrentActivity 1202
com.lispworks.BugFormLogsList 1202
com.lispworks.BugFormViewer 1203

44 The MP Package 1205
allowing-block-interrupts 1205
any-other-process-non-internal-server-p 1207
barrier-arriver-count 1208
barrier-block-and-wait 1208
barrier-change-count 1212
barrier-count 1212
barrier-disable 1213
barrier-enable 1214
barrier-name 1214
barrier-pass-through 1215
barrier-unblock 1216
barrier-wait 1217
change-process-priority 1220
condition-variable-broadcast 1221
condition-variable-signal 1221
condition-variable-wait 1222
condition-variable-wait-count 1224
*current-process* 1224
current-process-block-interrupts 1224
current-process-in-cleanup-p 1226
current-process-kill 1226
current-process-pause 1227
current-process-send 1229
current-process-set-terminate-method 1230
current-process-unblock-interrupts 1231
debug-other-process 1232
*default-process-priority* 1233
ensure-process-cleanup 1233
find-process-from-name 1235
funcall-async 1236
funcall-async-list 1236
general-handle-event 1237
get-current-process 1238
get-process 1239
get-process-private-property 1240
initialize-multiprocessing 1240
*initial-processes* 1242
last-callback-on-thread 1242
list-all-processes 1243
lock-and-condition-variable-broadcast 1244
lock-and-condition-variable-signal 1245
lock-and-condition-variable-wait 1246
lock-locked-p 1248
lock-owned-by-current-process-p 1248
lock-recursive-p 1249
lock-rewesively-locked-p 1250
lock-name 1250
lock-owner 1251
mailbox-count 1252
mailbox-empty-p 1253
mailbox-not-empty-p 1253
mailbox-peek 1254
mailbox-read 1255
mailbox-reader-process 1256
mailbox-send 1257
mailbox-wait 1257
mailbox-wait-for-event 1259
*main-process* 1261
make-barrier 1261
make-condition-variable 1262
make-lock 1263
make-mailbox 1265
make-named-timer 1266
make-semaphore 1267
make-timer 1268
map-all-processes 1269
map-all-processes-backtrace 1270
map-process-backtrace 1270
map-processes 1271
notice-fd 1272
process-alive-p 1272
process-all-events 1273
process-allow-scheduling 1274
process-arrest-reasons 1274
process-break 1275
process-continue 1275
process-exclusive-lock 1276
process-exclusive-unlock 1277
process-idle-time 1278
*process-initial-bindings* 1278
process-internal-server-p 1279
process-interrupt 1280
process-interrupt-list 1281
process-join 1282
process-kill 1283
process-lock 1284
process-mailbox 1285
process-name 1285
process-p 1286
process-plist 1286
process-poke 1287
process-priority 1289
process-private-property 1290
process-property 1291
process-reset 1292
process-run-function 1293
process-run-reasons 1296
process-run-time 1297
process-send 1298
process-sharing-lock 1299
process-sharing-unlock 1300
process-stop 1301
process-stopped-p 1302
process-terminate 1302
process-unlock 1304
process-unstop 1305
process-wait 1306
process-wait-for-event 1307
process-wait-function 1308
process-wait-local 1308
process-wait-local-with-periodic-checks 1311
process-wait-local-with-timeout 1313
process-wait-local-with-timeout-and-periodic-checks 1314
process-wait-with-timeout 1314
process-whostate 1316
processes-count 1316
pushnew-to-process-private-property 1317
pushnew-to-process-property 1318
ps 1319
remove-from-process-private-property 1320
remove-from-process-property 1320
remove-process-private-property 1321
remove-process-property 1322
schedule-timer 1323
schedule-timer-milliseconds 1325
schedule-timer-relative 1327
schedule-timer-relative-milliseconds 1328
semaphore-acquire 1330
semaphore-count 1331
semaphore-name 1332
semaphore-release 1332
semaphore-wait-count 1333
set-funcall-async-limit 1334
simple-lock-and-condition-variable-wait 1334
symeval-in-process 1335
timer-expired-p 1336
timer-name 1337
unnotice-fd 1339
unschedule-timer 1339
wait-processing-events 1340
with-exclusive-lock 1342
with-interrupts-blocked 1343
with-lock 1343
with-sharing-lock 1344
without-interrupts 1345
without-preemption 1346
yield 1347

45 The PARSERGEN Package 1349
defparser 1349

46 The SERIAL-PORT Package 1351
open-serial-port 1351
close-serial-port 1353
get-serial-port-state 1354
serial-port 1354
read-serial-port-char 1355
read-serial-port-string 1355
serial-port-input-available-p 1356
set-serial-port-state 1357
wait-serial-port-state 1357
write-serial-port-char 1358
write-serial-port-string 1359

47 The SQL Package 1361
add-sql-stream 1361
attribute-type 1362
cache-table-queries 1364
*cache-table-queries-default* 1365
commit 1365
connect 1366
*connect-if-exists* 1373
connected-databases 1374
create-index 1375
create-table 1376
create-view 1377
create-view-from-class 1378
database-name 1379
decode-to-db-standard-date 1380
decode-to-db-standard-timestamp 1380
encode-db-standard-date 1381
encode-db-standard-timestamp 1381
*default-database* 1382
*default-database-type* 1382
*default-update-objects-max-len* 1383
def-view-class 1383
delete-instance-records 1390
delete-records 1391
delete-sql-stream 1392
destroy-prepared-statement 1393
disable-sql-reader-syntax 1393
disconnect 1394
do-query 1395
Contents

ora-lob-file-get-name 1432
ora-lob-file-is-open 1432
ora-lob-file-open 1433
ora-lob-file-set-name 1434
ora-lob-flush-buffer 1435
ora-lob-free 1436
ora-lob-free-temporary 1437
ora-lob-get-buffer 1437
ora-lob-get-chunk-size 1440
ora-lob-get-length 1440
ora-lob-internal-lob-p 1441
ora-lob-is-equal 1442
ora-lob-is-open 1442
ora-lob-is-temporary 1443
ora-lob-load-from-file 1444
ora-lob-lob-locator 1445
ora-lob-locator-is-init 1446
ora-lob-open 1446
ora-lob-read-buffer 1447
ora-lob-read-into-plain-file 1450
ora-lob-read-foreign-buffer 1451
ora-lob-svc-ctx-handle 1452
ora-lob-trim 1452
ora-lob-write-buffer 1453
ora-lob-write-from-plain-file 1455
ora-lob-write-foreign-buffer 1456
p-oci-env 1457
p-oci-file 1458
p-oci-lob-locator 1458
p-oci-lob-or-file 1458
p-oci-svc-ctx 1458
prepare-statement 1459
print-query 1461
query 1462
reconnect 1464
restore-sql-reader-syntax-state 1465
rollback 1465
select 1466
set-prepared-statement-variables 1470
simple-do-query 1471
sql 1473
sql-connection-error 1474
sql-database-data-error 1474
sql-database-error 1475
*sql-enlarge-static* 1476
sql-expression 1476
sql-fatal-error 1477
*sql-libraries* 1477
*sql-loading-verbose* 1478
sql-operation 1478
sql-operator 1480
sql-recording-p 1481
sql-stream 1482
sql-temporary-error 1483
sql-timeout-error 1483
sql-user-error 1483
standard-db-object 1484
start-sql-recording 1484
status 1485
stop-sql-recording 1486
table-exists-p 1487
update-instance-from-records 1487
update-objects-joins 1488
update-records 1490
update-records-from-instance 1491
update-record-from-slot 1491
update-slot-from-record 1492
with-transaction 1493

48 The STREAM Package 1495

buffered-stream 1495
fundamental-binary-input-stream 1497
fundamental-binary-output-stream 1498
fundamental-binary-stream 1498
fundamental-character-input-stream 1499
fundamental-character-output-stream 1500
fundamental-character-stream 1501
fundamental-input-stream 1501
fundamental-output-stream 1502
fundamental-stream 1502
stream-advance-to-column 1503
stream-check-eof-no-hang 1504
stream-clear-input 1504
stream-clear-output 1505
stream-file-position 1506
stream-fill-buffer 1506
stream-finish-output 1507
stream-flush-buffer 1508
stream-force-output 1509
stream-fresh-line 1509
stream-line-column 1510
stream-listen 1511
stream-output-width 1512
stream-peek-char 1512
stream-read-buffer 1513
stream-read-byte 1514
stream-read-char 1515
stream-read-char-no-hang 1515
stream-read-line 1516
stream-read-sequence 1517
stream-read-timeout 1518
stream-start-line-p 1518
stream-terpri 1519
stream-unread-char 1520
stream-write-buffer 1520
stream-write-byte 1521
stream-write-char 1522
stream-write-sequence 1522
stream-write-string 1523
with-stream-input-buffer 1524
with-stream-output-buffer 1526

49  The SYSTEM Package  1529

apply-with-allocation-in-gen-num 1529
approaching-memory-limit 1530
atomic-decf 1531
atomic-incf 1531
atomic-exchange 1532
atomic-fixnum-decf 1532
atomic-fixnum-incf 1532
atomic-pop 1533
atomic-push 1534
augmented-string 1535
simple-augmented-string 1535
augmented-string-p 1535
simple-augmented-string-p 1535
*binary-file-type* 1536
*binary-file-types* 1536
call-system 1537
call-system-showing-output 1540
cdr-assoc 1543
*check-network-server* 1544
coerce-to-gesture-spec 1544
compare-and-swap 1546
copy-preferences-from-older-version 1547
count-gen-num-allocation 1548
*debug-initialization-errors-in-snap-shot* 1549
default-eol-style 1549
*default-stack-group-list-length* 1550
define-atomic-modify-macro 1551
define-top-loop-command 1552
detect-eol-style 1554
detect-japanese-encoding-in-file 1556
detect-unicode-bom 1556
detect-utf32-bom 1556
detect-utf8-bom 1556
*directory-link-transparency* 1558
ensure-loads-after-loads 1558
ensure-memory-after-store 1559
ensure-stores-after-memory 1560
ensure-stores-after-stores 1560
*extended-spaces* 1561
*file-encoding-detection-algorithm* 1562
file-encoding-resolution-error 1563
*file-eol-style-detection-algorithm* 1564
*filename-pattern-encoding-matches* 1564
find-encoding-option 1564
find-filename-pattern-encoding-match 1565
Contents

gen-num-segments-fragmentation-state 1566
generation-number 1568
gesture-spec-accelerator-bit 1568
gesture-spec-caps-lock-bit 1568
gesture-spec-control-bit 1568
gesture-spec-hyper-bit 1568
gesture-spec-meta-bit 1568
gesture-spec-shift-bit 1568
gesture-spec-super-bit 1568
gesture-spec-data 1570
gesture-spec-modifiers 1571
gesture-spec-p 1571
gesture-spec-to-character 1572
get-file-stat 1573
get-folder-path 1575
get-user-profile-directory 1578
guess-external-format 1579
immediatep 1580
in-static-area 1580
int32 1581
int32* 1582
int32+ 1582
int32- 1582
int32/ 1582
+int32-0+ 1583
+int32-1+ 1583
int32-1+ 1584
int32-1- 1584
int32/= 1585
int32< 1585
int32=< 1585
int32= 1585
int32> 1585
int32>= 1585
int32<< 1586
int32>> 1586
int32-aref 1587
int32-logand 1588
int32-logandc1 1588
int32-logandc2 1588
int64-lognor 1599
int64-lognot 1599
int64-logorc1 1599
int64-logorc2 1599
int64-logtest 1599
int64-logxor 1599
int64-minusp 1601
int64-plusp 1601
int64-zerop 1601
int64-to-int32 1602
int64-to-integer 1603
integer-to-int32 1603
integer-to-int64 1604
*line-arguments-list* 1604
load-data-file 1605
locale-file-encoding 1605
low-level-atomic-place-p 1606
make-gesture-spec 1607
make-simple-int32-vector 1612
make-simple-int64-vector 1613
make-stderr-stream 1614
make-typed-aref-vector 1615
map-environment 1616
marking-gc 1617
memory-growth-margin 1619
merge-ef-specs 1620
object-address 1621
object-pointer 1622
octet-ref 1623
base-char-ref 1623
open-pipe 1624
open-url 1629
package-flagged-p 1629
pipe-exit-status 1630
pipe-kill-process 1631
pointer-from-address 1633
print-pretty-gesture-spec 1634
*print-symbols-using-bars* 1635
product-registry-path 1636
room-values 1637
run-shell-command 1638
safe-locale-file-encoding 1643
set-approaching-memory-limit-callback 1644
set-automatic-gc-callback 1645
set-blocking-gen-num 1647
set-default-segment-size 1649
set-delay-promotion 1651
set-file-dates 1652
set-gen-num-gc-threshold 1653
set-maximum-memory 1654
set-maximum-segment-size 1655
set-memory-check 1657
set-memory-exhausted-callback 1658
set-signal-handler 1660
set-spare-keeping-policy 1661
set-temp-directory 1662
setup-atomic-funcall 1663
*sg-default-size* 1664
simple-int32-vector 1665
simple-int64-vector 1665
*stack-overflow-behaviour* 1666
staticp 1667
storage-exhausted 1668
sweep-gen-num-objects 1668
typed-aref 1669
wait-for-input-streams 1671
wait-for-input-streams-returning-first 1672
with-modification-change 1673
with-modification-check-macro 1674
with-other-threads-disabled 1675

50 Miscellaneous WIN32 symbols 1677

canonicalize-sid-string 1677
connect-to-named-pipe 1678
dismiss-splash-screen 1680
impersonating-named-pipe-client 1681
impersonating-user 1682
known-sid-integer-to-sid-string 1683
*latin-1-code-pages* 1684
Contents

long-namestring 1685
*multibyte-code-page-ef* 1686
named-pipe-stream-name 1686
open-named-pipe-stream 1687
record-message-in-windows-event-log 1690
security-description-string-for-open-named-pipe 1692
set-application-themed 1696
short-namestring 1697
sid-string-to-user-name 1698
str 1698
lpCTSTR 1698
lpstr 1698
tstr 1699
lpcCTSTR 1699
lpcTSTR 1699
wstr 1700
lpcWSTR 1700
lwstr 1700
user-name-to-sid-string 1701
wait-for-connection 1701
with-windows-event-log-event-source 1702

51 The Windows registry API  1705

close-registry-key 1705
collect-registry-subkeys 1706
collect-registry-values 1707
create-registry-key 1709
delete-registry-key 1710
enum-registry-value 1711
open-registry-key 1714
query-registry-key-info 1715
query-registry-value 1716
registry-key-exists-p 1717
registry-value 1717
set-registry-value 1719
with-registry-key 1720

52 The DDE client interface  1723

dde-advise-start 1723
53 The DDE server interface 1753

dde-server-poke 1753
dde-server-request 1754
dde-server-topic 1755
dde-server-topics 1756
dde-system-topic 1757
dde-topic 1757
dde-topic-items 1758
declare-dde-dispatch-topic 1759
declare-dde-server 1760
declare-dde-server-function 1761
start-dde-server 1765

54 Dynamic library C functions 1767

InitLispWorks 1767
LispWorksDlsym 1770
LispWorksState 1771
SimpleInitLispWorks 1772
Contents

QuitLispWorks 1773

Index 1775
Preface

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

User Guide section

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

These chapters describe the central programming tools and features in Lisp-
Works:

• 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) lis-
tener.
• 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”.

xlvii
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, “Storage 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”.

xlviii
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 39, “The LINK-LOAD Package”, describes symbols available in the LINK-LOAD package. It applies to LispWorks for UNIX only (not LispWorks for Linux, FreeBSD, Mac OS X or x86/x64 Solaris).
- 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.

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 Foreign Language Interface User Guide and Reference Manual* explains how you can use C source code in applications developed using LispWorks.

• 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](http://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


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/7-0-0-0/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")
```
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 Linux, FreeBSD and UNIX 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 150 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-7-0-0-macos-universal -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-7-0-0-x86-win32.exe -build C:\temp\save-config.lisp
```

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

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

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

```
% lispworks-7-0-0-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 392

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 149.

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:

```
(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 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 \texttt{print}. You see it here because output sent to \texttt{*standard-output*} is written to the listener. The second ‘42’ printed is the return value of the call to \texttt{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:

\begin{verbatim}
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)
\end{verbatim}

\section*{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.

\subsection*{2.2.1 Standard top-level loop commands}

\texttt{:redo} \hspace{1cm} \textit{Listener command}

\begin{verbatim}
:redo &optional command-identifier
\end{verbatim}

This option repeats a previous input. The \textit{command-identifier} is either a number in the listener’s history list or a symbol or subform in the input to repeat. If \textit{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
:?
:? 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 1022 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.

```
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)
9
\end{verbatim}

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 573 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:
The Debugger

CL-USER 10 > (defun function-1 (a b c)
  (function-2 (+ a b) c))
FUNCTION-1

CL-USER 11 > (defun function-2 (a b)
  (function-3 (+ a b)))
FUNCTION-2

CL-USER 12 > (defun function-3 (a) (/ 3 (- 111 a)))
FUNCTION-3

CL-USER 13 > (function-1 1 10 100)

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 14 : 1 > :bq 10

SYSTEM::DIVISION-BY-ZERO-ERROR <- / <- FUNCTION-3
  <- SYSTEM::*%APPLY-INTERPRETED-FUNCTION <- FUNCTION-2
  <- SYSTEM::*%APPLY-INTERPRETED-FUNCTION <- FUNCTION-1
  <- SYSTEM::*%APPLY-INTERPRETED-FUNCTION <- SYSTEM::%INVOKE <-
  SYSTEM::%EVAL

CL-USER 15 : 1 >

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.

:b
  Debugger command

:b &optional verbose m

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.

\[\text{bg} \quad \text{Debugger command} \]

\[\text{bg} \quad m \]

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.

\[> \quad \text{Debugger command} \]

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

\[< \quad \text{Debugger command} \]

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

\[p \quad \text{Debugger command} \]

\[p \quad m | fn-name | fn-name-substring \]

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

[:l]

: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 \texttt{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 \texttt{var-name-substring} then it prints the value of the first variable whose name contains \texttt{var-name-substring}.

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

**:error** \hspace{1cm} \textit{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** \hspace{1cm} \textit{Debugger command}

**:cc \&optional var**

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

**:ed** \hspace{1cm} \textit{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** \hspace{1cm} \textit{Debugger command}

**:all \&optional flag**

This option enables you to set the debugger option to show all frames (if \texttt{flag} is non-nil), or back to the default (if \texttt{flag} is nil). By default, \texttt{flag} is \texttt{t}.

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

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

12. When prompted for a form to be evaluated, enter \texttt{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 :\texttt{preserve} or :\texttt{down-case}.

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

\begin{verbatim}
(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))
\end{verbatim}

3.6 Debugger control variables

\texttt{cl:*debug-io*}

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

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

`dbg:*debug-print-level*`

The value to which `cl:*print-level*` is bound during output from the debugger.

`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).

`dbg:*print-binding-frames*`

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

`dbg:*print-catch-frames*`

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

`dbg:*print-handler-frames*`

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

`dbg:*print-restart-frames*`

This variable controls whether restart frames are displayed by the debugger. The initial value is `nil`. The value can be set directly or by calling `set-debugger-options` which may be more convenient.
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>:i m</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 28.</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 *inspect-print-level* and
*inspect-print-length* are similar to *describe-print-level* and
*describe-print-length* (see above).

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

`(\text{:dm} \text{ or } \text{:dr for more})`

If you enter the command :dm at the prompt it displays the next *describe-length* slots, and if you enter :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, \texttt{:dm} does not do anything useful. Typing \texttt{:d} lets you view the object again.

\texttt{:ud} is equivalent to typing \texttt{:u} followed by \texttt{:d}.

### 4.3 Inspection modes

The \texttt{: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 \texttt{: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))
*HASH*

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)

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:

```
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
```
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

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 45. 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 \texttt{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, \texttt{trace} prints the following information:

- The level of tracing, that is, the number of recursive entries to \texttt{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, \texttt{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
CL-USER 7 >
```

### 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:
   
   (setq args-in-reverse ())

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

3. In the listener, evaluate the following form:
   
   (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.

:after 

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

:eval-before 

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 Tracing options

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

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

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

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

Generates a backtrace on each call to the traced function. backtrace 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.

5.2.4 Entering stepping mode

:step

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

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

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:
   
   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:
   
   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

Trace keyword

:when form

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

5.2.8 Storing the memory allocation made during a function call

:allocation

Trace keyword

:allocation form

If form is non-nil, this prints the memory allocation, in bytes, made during a function call. The symbol that 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 fac function, set the value of :allocation to $$fac-alloc.
2. In the listener, call fac, and then evaluate $$fac-alloc.

   CL-USER 152 > $$fac-alloc
   744

5.2.9 Tracing functions from inside other functions

:inside

Trace keyword

:inside list-of-functions

The functions given in the argument to :inside should reference the traced function in their implementation. The traced function is then only
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.
   
   CL-USER 2 > (fac 3)
   6

4. Call **fac2**, and notice the tracing information.
   
   Evaluate `(fac2 3)`, and notice the tracing information.
   
   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 argu-
ment 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 follow-
ing 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 *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 *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))

(defun 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 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.

5.6 Troubleshooting tracing

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

5.6.1 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.
5.6.2 Missing output

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

5.6.2.1 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.

5.6.2.2 trace works on function names, not function objects

\texttt{trace} works by tracing function names, not function objects. Therefore tracing function objects, for example by

\begin{verbatim}
(trace #\'foo)
\end{verbatim}

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

\begin{verbatim}
(trace foo)
\end{verbatim}

Also, if the symbol \texttt{foo} is traced, then code which invokes \texttt{foo} by

\begin{verbatim}
(funcall (symbol-function \'foo) ...)  
\end{verbatim}

or equivalently

\begin{verbatim}
(funcall #\'foo ...)
\end{verbatim}

will not produce any trace output.

The correct approach is to use \texttt{(funcall \'foo ...)} instead of \texttt{(funcall #\'foo ...)}.
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:

```
(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:

```
(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 48 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 960 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:

```
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:
(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
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 defining 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 \texttt{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).
Note that \texttt{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 \texttt{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 \texttt{defadvice} by giving it a \texttt{:where} keyword with value \texttt{:end}. 
Examples

CL-USER 41 > (defadvice
   (alpha zero-or-negative :around :where :end)
   (x)
   (unless (plusp x)
     (format t "**Warning: alpha is being called with ~A" x))
   (if (zerop x)
     (error "Alpha cannot be called with zero")
     (call-next-advice x)))
NIL

CL-USER 42 > (alpha -5 -2)
**Warning: alpha is being called with -5
**Warning: alpha is being called with -4
**Warning: alpha is being called with -3
**Warning: alpha is being called with -2
1669/3600

CL-USER 43 > (alpha 0 3)
**Warning: alpha is being called with 0
Error: alpha cannot be called with zero
1 (abort) return to level 0.
2 return to top loop level 0

Type :c followed by a number to proceed

CL-USER 44 : 1 > :a

Finally you decide to alter alpha yet again, this time to produce approximations to \(\pi\). \(\frac{\pi^2}{6}\) 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.)

CL-USER 51 > (defadvice
   (alpha pi-approximation :around)
   (limit)
   (sqrt
    (* 6
     (call-next-advice 1.0 limit))))
NIL
Next, try calling the following in turn:

```
(alpa 10.0)
(alpa 100.0)
(alpa 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)))

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

### 6.6 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 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 dspecs:

    (car)
    (setf car)
(defclass standard-object)

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.

`(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:

`(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:

`(defmethod name qualifications (specializer*))`

name := symbol | (setf symbol)
qualifications := qualifier | (qualifier qualification*)
qualifier := symbol
specializer := symbol | (eql object)
Functions in the dspec API accept non-canonical dspecs. All dspec functions, except `dspec:prettify-dspec`, `find-dspec-locations`, `name-definition-locations`, `dspec-definition-locations` and `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 system dspec classes and aliases” on page 66 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 `dspec-defined-p` and `dspec-undefiner` for testing if a dspec is currently defined and for undefining a dspec.

New dspec classes are defined using `define-dspec-class`.

Dspec classes can be subclassed. The top-level classes correspond to distinct global namespaces (such as `variable` for variables and constants and `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 system dspec classes and aliases” on page 66 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 64 for how to add new ways of making existing definitions.

7.3.1.1 Complete example of a top-level dspec class

Define a `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
(dschange:define-dspec-class def-saved-value nil
  "Defined saved values"
  :definedp
  '(
    (lambda (name)
      ;; Find any object that def-saved-value recorded
      (not (null (find-saved-value name)))))
  :undefineder
  '(
    (lambda (dspec)
      ;; Remove what def-saved-value recorded
      ~(remprop , (dschange:dspec-name dspec) 'saved-value))
    (lambda (obj)
      ;; Given a saved-value object, we can reconstruct its dspec
      ~(def-saved-value ,(saved-value-name obj)))))
```

For completeness, define a form parser that generates dspecs from forms:

```lisp
(dschange:define-form-parser
  (def-saved-value
    {(:parser dschange:single-form-form-parser)}))
```
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 61.

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

```lisp
(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:

```lisp
(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 function to compute a computed-saved-value:

```lisp
(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:

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

Define a dspec class for def-computed-saved-value dspecs:
(dspec:define-dspec-class def-computed-saved-value def-saved-value
  "Defined computed saved values"
  :definedp
  #'(lambda (name)
    ;; Find any object that def-computed-saved-value recorded
    (computed-saved-value-p (find-saved-value name)))
  ;; The :undefiner is inherited from the superspace.
  :object-dspec
  #'(lambda (obj)
    ;; Given a computed-saved-value object, we can reconstruct its dspec
    (and (computed-saved-value-p obj)
      `(def-computed-saved-value ,(saved-value-name obj))))
)

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:

(dspec: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.
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

```lisp
(defun parameterdef (value name)
  `(defparameter ,name ,value))
```

then

```lisp
(dsdp: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.

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 73.
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 system dspec classes and aliases

This section shows the dspec classes, subclasses and aliases provided by the system. 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 system dspec classes and aliases

COMPILER-MACRO (alias DEFINE-COMPILER-MACRO)
EDITOR:DEFINE-COMMAND (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 DEF METHOD)
METHOD-COMBINATION (alias DEFINE-METHOD-COMBINATION)
PACKAGE (alias DEFPACKAGE)
STRUCTURE (alias DEFSTRUCT)
TYPE
DEFCLASS
CAPI:DEFINE-INTERFACE
CAPI:DEFINE-LAYOUT
DEFINE-CONDITION
STRUCTURE-CLASS
DEFTYPE
VARIABLE
DEFINE-SYMBOL-MACRO
DEFCONSTANT
DEFVAR (aliases DEFGLOBAL-PARAMETER, DEFGLOBAL-VARIABLE, DEFPARAMETER)

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 CLOS dspec classes

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

7.5.2 Part Classes

method is a part class for defgeneric.

compiler-macro is a part class for function.

7.5.3 Foreign callable dspecs

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

7.6 Subfunction dspecs

For some purposes, we allow dspecs that do not name a global definition, but a local function. These are of the form

(subfunction name parent)

where parent is another dspec (possibly even a subfunction dspec).

name is a symbol, a list, or a number, but it is not used for anything within the dspec system. A subfunction dspec can be canonicalized and prettified, and 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 <#>))

location is a basic location and <#> 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:

- **Pathname**: A definition existed in the file named or an editor buffer with that name.
- **listener**: A definition was executed interactively in the listener or an editor buffer not associated with a file.
- **unknown**: A definition was found in the image (these are entered when a location query does not find any information already in the database).
- **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.
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 696 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 61.

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

```lisp
(dspec: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

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

1. Load a file `foo.lisp` containing
   ```lisp
   (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 64), like this:

```
(dsparse:define-form-parser (my-defun
   (:parser dsparse: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".
7.9.4 Reusing form parsers

The form parser established above was specifically for \texttt{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:

\begin{verbatim}
(dspez:define-form-parser (name-second (:anonymous t))
  (value name)
  `,(name-second ,name))
\end{verbatim}

Note that the \texttt{name-second} variable is evaluated in the body of the parser. Supposing you have another defining macro \texttt{constantdef}:

\begin{verbatim}
(defmacro constantdef (value name)
  `(defconstant ,name ,value))
\end{verbatim}

then you can associate the same parser with both this and \texttt{parameterdef}:

\begin{verbatim}
(dspez:define-form-parser (parameterdef
  (:parser name-second-form-parser)))
(dspez:define-form-parser (constantdef
  (:parser name-second-form-parser)))
\end{verbatim}

7.9.5 Example: defcondition

Suppose you have a macro based on \texttt{define-condition}:

\begin{verbatim}
(defmacro defcondition (&rest args)
  `(define-condition ,@args))
\end{verbatim}

When the following form is evaluated, the system records the dspez (define-condition foo):

\begin{verbatim}
(defcondition foo () ())
\end{verbatim}

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:

\begin{verbatim}
(dspez:define-form-parser
  (defcondition
   (:alias define-condition)))
\end{verbatim}

So now:
7.9 Users of location information

\( (\text{dspec:parse-form-dspec} \ '(\text{defcondition foo} () ())) \)
\( => \)
\( (\text{defcondition foo}) \)

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 ...".

\( (\text{dspec:define-dspec-alias dspec-alias defcondition (name)} \n  \text{`\text{(define-condition ,name)}}) \)

So now this definition can be located:

\( (\text{defcondition foo} () ()) \)

just as if it were:

\( (\text{define-condition foo} () ()) \)

7.9.6 Example: my-defmethod

Suppose you have a method definer `my-defmethod`:

\( (\text{defmacro my-defmethod ((name &key doc)} \n  \text{lambda-list} \n  \text{&body body)} \n  \text{`\text{(defmethod ,name ,lambda-list ,@body)}}) \)

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:

\( (\text{dspec:define-dspec-alias my-defmethod (name &rest options)} \n  \text{`\text{(defmethod ,name ,lambda-list ,@options)}}) \)

The dspec parser for method lambda lists is complicated, but you can invoke the `defmethod` parser in your `my-defmethod` parser, like this:

\( (\text{dspec:define-form-parser my-defmethod (name-stuff lambda-list)} \n  \text{`\text{(defmethod ,,(my-defmethod ,@(cdr (dspec:parse-form-dspec \n    \text{`\text{(defmethod ,(car name-stuff)} \n    ,lambda-list))))}}) }}) \)
Now this definition can be located:

```
(my-defmethod (bar :doc "bar documentation") (x y)
  (foo x y))
```

just as if it were:

```
(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.
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 undefine-action-list, print-actions, execute-actions, define-action and undefine-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")
```
8 Action Lists

(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”.

(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-
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 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)
                     \n                     (dotimes (n a) (funcall b)))))
; FRED
FRED
NIL
NIL

CL-USER 4 > (funcall (compile nil '(lambda (n)
                     \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 91 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>
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>debug=3</td>
<td>Avoids an optimization to last</td>
</tr>
<tr>
<td>safety&gt;1</td>
<td>Be careful when multiple value counts are wrong</td>
</tr>
<tr>
<td>safety&lt;1</td>
<td>Do not check array indices during write</td>
</tr>
<tr>
<td>safety&lt;2</td>
<td>Do not check array indices during read</td>
</tr>
<tr>
<td>speed&gt;space</td>
<td>Inline map functions (unless debug&gt;2)</td>
</tr>
<tr>
<td>debug&lt;=2</td>
<td>Optimize (merge) tail calls</td>
</tr>
<tr>
<td>debug&lt;2 and safety&lt;2</td>
<td>Self calls</td>
</tr>
<tr>
<td>safety&gt;=2</td>
<td>Check get special</td>
</tr>
<tr>
<td>safety&lt;2</td>
<td>Do not check types during write</td>
</tr>
<tr>
<td>safety&lt;3</td>
<td>Do not check types during read</td>
</tr>
<tr>
<td>safety&gt;=1</td>
<td>Check structure access</td>
</tr>
<tr>
<td>safety&lt;1</td>
<td>Inline structure readers, with no type check</td>
</tr>
<tr>
<td>safety=0</td>
<td>Inline structure writers, with no type check</td>
</tr>
<tr>
<td>safety&gt;1</td>
<td>Check number of args</td>
</tr>
<tr>
<td>safety&gt;=1 or interruptible&gt;0</td>
<td>Check stack overflow</td>
</tr>
<tr>
<td>safety&gt;1</td>
<td>Ensures the thing being funcalled is a function</td>
</tr>
<tr>
<td>safety&lt;3 and fixnum-safety=2</td>
<td>Fixnum-only arithmetic with errors for non fixnum arguments.</td>
</tr>
<tr>
<td>safety&lt;3 and fixnum-safety=1</td>
<td>No fixnum overflow checks</td>
</tr>
<tr>
<td>safety&lt;3 and fixnum-safety=0</td>
<td>No fixnum arithmetic checks at all</td>
</tr>
<tr>
<td>safety&gt;2</td>
<td>char = checks for arguments of type character</td>
</tr>
<tr>
<td>safety&gt;=2</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:

```
(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:

```
(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:

```
(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:

```lisp
(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:

```lisp
(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 dec-
larations 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 analyse 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)))
```

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 410 and “Fast 64-bit arithmetic” on page 412.

### 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:

```lisp
(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.

Whensetq is 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 is 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 speeded up by declarations of the symbol, normally by using proclaim or declaim.

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 runtime it also signals an error.
  
  In non-SMP LispWorks the compiler gives an error, but there is no runtime check. The runtime 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 `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.

### 9.7.5.1 Finding symbols to declare

The macro `analysing-special-variables-usage` can be used to find symbols that may be proclaimed global, which can improve performance. `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

`(declare dynamic-extent)` will optimize these calls so that they allocate in the stack, in all cases:

- `&rest` lists
- `flet` functions and `labels` functions
- `(cons x y)`
- `(list ...)`
- `(list* ...)`
- `(copy-list x)`
- `(make-list x)`
- `(vector ...)`
The Compiler

(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 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 runtime, as described in Chapter 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 103.
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` (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
(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 colored 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.
11

Storage Management

This chapter introduces some basic ideas of storage management, and then discusses the LispWorks storage management system in more detail. The chapter also introduces the functions and macros needed to control storage 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.
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 396.

Garbage collection with a naive algorithm is extremely inefficient.

The LispWorks GC works in unison with the storage 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 storage management system

11.2.1 General guidance

The storage 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 storage management, you should be familiar with the function room, and use it frequently. There is no point at all in trying to tune the storage management without knowing the sizes of your application, as output by room.
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 storage 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 storage 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](http://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, 100-200Mb 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 100-200Mb 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 \texttt{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 \texttt{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 col-
lected automatically, so there is a good chance that you do not have to do any-
thing. However, you may want to call \texttt{gc-generation} explicitly when you
know it is a good time to do it. You may also want to block automatic calls if
they they take too long: use \texttt{set-blocking-gen-num} to do that. If generation
3 becomes very big (Gigabytes), you may also consider using \texttt{marking-gc}
instead of \texttt{gc-generation}.

Once you set up \texttt{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. On 32-bit LispWorks you can use \texttt{check-fragmenta-
tion} to check for fragmentation, and \texttt{try-move-in-generation} to prevent it
if needed. See “Controlling Fragmentation” on page 126 for a discussion.

In 64-bit LispWorks you have a problem with fragmentation only if you use
\texttt{marking-gc} \texttt{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. \texttt{gc-generation} can be used occasionally to
eliminate all fragmentation. Check for fragmentation by using \texttt{gen-num-seg-
ments-fragmentation-state}.

\subsection{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 \texttt{clean-down}, which is called automatically (by default) when saving an
image (whether by \texttt{save-image} or \texttt{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 7.0.

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 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 UNIX only (not LispWorks for Linux, x86/x64 Solaris, FreeBSD or Macintosh).

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-genera-
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 133 and “Allocation of stacks” on page 134.

### 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 \texttt{minimum-for-sweep} value, which is set by \texttt{set-gc-parameters}. If the amount allocated is less than \texttt{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 \texttt{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 \texttt{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 \texttt{set-minimum-free-space}, and can be shown by \texttt{(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 \texttt{minimum-for-promote}, which is set by \texttt{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 (0.5GB). LispWorks then tries to locate the location of dynamic libraries, and marks a region around these libraries that should not be used (by default 64MB 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 398.

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 2.5GB), 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 7.0.

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 398.
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 \texttt{ReserveSize}.

Normally the dynamic libraries are mapped at \texttt{#x28000000}, and therefore LispWorks can grow without a problem.

In principle LispWorks can grow up to some distance below \texttt{#xC0000000} (almost 2.25GB), though this depends on the OS kernel allowing this size and how many threads you have running.

\subsection*{11.3.5.3 \textit{x86/x64 Solaris}}

The default initial heap is mapped at address \texttt{#x10000000} (0.25GB). LispWorks then tries to locate the location of dynamic libraries, and marks a region around these libraries that should not be used (by default 64MB from the bottom). In most cases this suffices to avoid clashes.

Problems can arise if the memory at \texttt{#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 398.

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 \texttt{ReserveSize}.

\subsection*{11.3.5.4 Windows and Macintosh}

LispWorks (32-bit) for Windows and LispWorks (32-bit) for Macintosh both map by default at \texttt{#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 \texttt{#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 398.

LispWorks (32-bit) for Windows can in principle grow up to some distance below \texttt{#x80000000} (almost 1.5GB) but there is always the possibility that
some DLL will be mapped in this region. On startup, it reserves 0.5GB above its location, so that much is guaranteed.

LispWorks (32-bit) for Macintosh can grow to around 2.7GB. You can relocate it only on the Intel architecture.

### 11.3.5.5 AIX

The image is mapped at \#x31000000.

The 16MB (= \#x10000000) 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 256MB between \#x20000000 and \#x30000000.

The heap is relocatable as described in “Startup relocation of 32-bit Lisp-Works” on page 398. 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](http://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 122.
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 \texttt{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 \texttt{approaching-memory-limit} (which is a subclass of \texttt{cl:storage-condition}).

The function \texttt{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 \texttt{: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 \texttt{:minimum-overflow} and \texttt{:maximum-overflow}, which can be set by \texttt{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 \texttt{: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 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 :new-generation-size (set by set-gc-parameters) in words (that is, multiples of 4 bytes), or the amount of free space needed, whichever is bigger. If :new-generation-size is 0, it is not expanded. In this case part of the objects marked for promotion are not promoted.

### 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 :new-generation-size multiplied by two. Garbage collection of generation 2 can be caused by calling the function collect-generation-2 with appropriate argument.

### 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 enlarge-generation to expand the generation in advance.

### 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 room or room-values and generation-number to follow the behavior of objects). In this case, 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. try-move-in-generation or 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
cases, \texttt{try-move-in-generation} is better than \texttt{try-compact-in-generation}.

The actual decision to use these functions will be typically based on the results of \texttt{check-fragmentation}. For example, the following function checks whether there is more than 10Mb free area in generation 2 in blocks of 4096 bytes or larger (tlb, third return value of \texttt{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 \texttt{try-move-in-generation}. Because \texttt{try-move-in-generation} gets a \texttt{time-threshold} of 0, it returns after moving at most one segment. (It will not move any segments if none of them looks fragmented.)

\begin{verbatim}
(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))))
\end{verbatim}

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 \texttt{try-move-in-generation} or \texttt{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 133. For full details of these functions, see their reference entries.

11.3.12.1 Determining storage usage

To determine storage usage (useful when benchmarking), use the functions \texttt{room}, \texttt{total-allocation} and \texttt{find-object-size}. The function \texttt{room-values} is suitable for programmatic use: it returns the values that \texttt{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 allocation 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).
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:

:other The segment contains a mixture of :other, :function and :symbol, but not :cons or :non-pointer.

:mixed The segment contains cons objects that are static.

:mixed-static The segment contains a mixture like :mixed, but static.

:non-pointer-static The segment contains objects that do not contain pointers and are static (currently stacks are also allocated in these segments).

: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 256MB, 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 131.

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 \texttt{other-big} and \texttt{: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 \texttt{extended-time} and periodical calls to \texttt{room}. 
In the output of \texttt{room} (or the more verbose \texttt{room t}), the allocation in each generation is presented according to the allocation type, which may be useful to decide on possible tuning.

\texttt{(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 \texttt{gc-generation}) or a marking garbage collection (calls to \texttt{marking-gc}).

In addition to \texttt{room} and \texttt{extended-time}, there are also the functions \texttt{count-gen-num-allocation}, \texttt{gen-num-segments-fragmentation-state}, and \texttt{set-automatic-gc-callback}. These function 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 \texttt{set-blocking-gen-num} and \texttt{set-gen-num-gc-threshold}. It may also be useful to set the maximum segment size with \texttt{set-maximum-segment-size}.

Explicit garbage collection can be done by calling \texttt{gc-generation} and \texttt{marking-gc}. Since repeated use of \texttt{marking-gc} will cause a lot of fragmentation, the arguments \texttt{what-to-copy} and \texttt{max-size-to-copy} can be used to specify that part of the data should be collected by copying.

\texttt{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 Common Memory Management Features

This section summarizes Memory Management functionality common to all LispWorks 7.0 implementations.

11.5.1 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.

11.5.2 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 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.

11.5.3 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.

### 11.5.4 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.

### 11.5.5 Mapping across all objects

To call a function on all objects in the image, use `sweep-all-objects`.

### 11.5.6 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-special-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.

### 11.5.7 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.*
11.5.8 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 880.

11.5.9 Assisting the garbage collector

This section describes techniques that may improve the performance of your application by reducing the GC’s workload.

11.5.9.1 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.
(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)))
    (setf (queue-tail 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:

(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 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. 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.

- It is possible to keep track of every function called during a particular computation, but significant effort is involved in determining which symbols are suitable for profiling and in keeping track of the results. To minimize this effort you need to specify which symbols to profile, either by naming the required symbols, or by naming a package, all of whose symbols are profiled. The profiler first checks that these symbols have indeed got function definitions and are therefore suitable for profiling.

- 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.

- On Unix/Linux/FreeBSD 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.

Below is an example of setting up the profiler:

```
(set-up-profiler :symbols '(car cdr) :style :list)
```

Here the functions car and cdr are going to be profiled and the output will be just the columnar report.

The function set-up-profiler adds symbols to the *profile-symbol-list*. The functions add-symbol-profiler and remove-symbol-profiler can also be used to change the symbols profiled.

The function set-profiler-threshold can be used with reset-profiler to control the effects of repeated profiler runs.

## 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 140. 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 140.

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>seen (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: MOD</td>
<td>17 ( 3)</td>
</tr>
<tr>
<td>2: FLOOR</td>
<td>5 ( 1)</td>
</tr>
<tr>
<td>1: EQL</td>
<td>8 ( 1)</td>
</tr>
<tr>
<td>1: &gt;=</td>
<td>7 ( 1)</td>
</tr>
<tr>
<td>2: REALP</td>
<td>2 ( 0)</td>
</tr>
<tr>
<td>1: +</td>
<td>6 ( 1)</td>
</tr>
<tr>
<td>1: LENGTH</td>
<td>4 ( 1)</td>
</tr>
</tbody>
</table>

Cumulative profile summary

<table>
<thead>
<tr>
<th>Symbol</th>
<th>called</th>
<th>profile (%)</th>
<th>top (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOD</td>
<td>1000000</td>
<td>17 ( 3)</td>
<td>8 ( 1)</td>
</tr>
<tr>
<td>EQL</td>
<td>2000117</td>
<td>8 ( 1)</td>
<td>8 ( 1)</td>
</tr>
<tr>
<td>&gt;=</td>
<td>1000001</td>
<td>7 ( 1)</td>
<td>5 ( 1)</td>
</tr>
<tr>
<td>+</td>
<td>1000000</td>
<td>6 ( 1)</td>
<td>6 ( 1)</td>
</tr>
<tr>
<td>FLOOR</td>
<td>1000000</td>
<td>5 ( 1)</td>
<td>5 ( 1)</td>
</tr>
<tr>
<td>LENGTH</td>
<td>2000086</td>
<td>4 ( 1)</td>
<td>4 ( 1)</td>
</tr>
<tr>
<td>REALP</td>
<td>1000001</td>
<td>2 ( 0)</td>
<td>2 ( 0)</td>
</tr>
</tbody>
</table>

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`.

### 12.5 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.

### 12.6 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:

```lisp
(member 'x '(x y z) :test #'eq)
```

is transformed to:

```lisp
(memq 'x '(x y z))
```

by the compiler and therefore the function `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 `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.

It is possible to profile all the symbols in the system by setting up the profiler as follows:

```
(set-up-profiler :package (list-all-packages))
```

### 12.7 Profiling and garbage collection

The macro `extended-time` provides useful information on garbage collection activities.

The `gc` argument of `set-up-profiler` controls whether or not the system’s memory management functions are profiled.
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 147 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 146 describes these initialization files.
13.3 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 startup. 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 HOMEPATH. 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 150 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:
13 Customization of LispWorks

```lisp
(in-package "CL-USER")
(load-all-patches)
(load #:mswindows "/tmp/my-configuration.lisp"
     #:mswindows "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 7.0/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:
  ```bash
  mymac$ "/Applications/LispWorks 7.0/LispWorks.app/Contents/MacOS/lispworks-7-0-0-macos-universal" -build save-image.lisp
  ``
  Your new application bundle is saved in `/Applications/LispWorks 7.0/My LispWorks.app`

- **On Microsoft Windows**, run in a MS-DOS window:
  ```cmd
  C:\temp\"C:\Program Files\LispWorks\lispworks-7-0-0-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:
  ```bash
  linux:/tmp$ lispworks-7-0-0-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 146.

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:

```lisp
(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-7-0-0-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 deliver with the :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 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 151 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 \textit{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 $\ast$, $\ast\ast$ and $\ast\ast\ast$.

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 \texttt{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 \texttt{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 \texttt{destroy-callback}, this information can be preserved over a call to \texttt{save-current-session} by defining methods on the generic functions \texttt{capi:interface-preserving-state-p} or \texttt{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 \textit{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:

\begin{verbatim}
(defvar *my-files*
  '('/path/to/foo1'
    '/path/to/foo2'
    '/path/to/foo3'))

(dolist (file *my-files*)
  (compile-file file :load t))
\end{verbatim}
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:

```
(setq editor:*maximum-ordinary-windows* 1)
```
This variable is set in the file `a-dot-lispworks.lisp`.

13.6.3 Binding commands to keystrokes
You can bind existing editor commands to different keystrokes, using `editor:bind-key`.

The LispWorks file `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.

```
(editor:bind-key "Self Insert" #\? :mode "Echo Area")
```
Since `?` is then no longer available for help, you may wish to rebind help to `Ctrl+?`.

```
(editor:bind-key "Help on Parse" "Control-?" :mode "Echo Area")
```
If you use another editor emulation, then see the LispWorks file `config/msw-key-binds.lisp` or `config/mac-key-binds.lisp` for the corresponding `editor:bind-key` forms.

13.7 Finding source code
Note: This section does not apply to LispWorks Personal Edition.

To configure LispWorks so that editor commands such as `Find Source`, the menu command `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:

```
(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 153 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 146) 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:

```lisp
(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:

```lisp
(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.

```lisp
(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 Unix/Linux/FreeBSD platforms.

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 conservative 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 32-bit LispWorks to build a dynamic library on Microsoft Windows, Intel Macintosh, Linux, x86/x64 Solaris and FreeBSD, and 64-bit LispWorks on Windows, Intel Macintosh, Linux and x86/x64 Solaris.

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 thenGetProcAddress. 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:

- **Linux** Link with *libpthread.so*
- **FreeBSD** Link with *libpthread.so*
- **Mac OS X** No special requirements
- **Solaris** Compile and link multithreaded (for example, using the `-mt` option to Oracle’s `cc`)

### 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:

```
------------- hello.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-7-0-0-x86-win32.exe -build hello.lisp
```

on the command line, or use the Application Builder tool.

(See “Saving a LispWorks image” on page 147 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 396.

### 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.

- Limited integration with CLOS.

- Make socket streams using Java sockets. See “Socket streams with Java sockets and SSL on Android” on page 369.

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>
<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>
<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.

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 jobject 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 Calling from Lisp to Java

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:

```
(import-java-class-definitions "java.io.File")
```

And to write the definitions to a file:

```
(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 170.

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 definition 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 that 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:

- **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**
  
  Defines a caller that calls a Java constructor method.

- **define-field-accessor**
15.2 Calling from Lisp to Java

Defines callers to access (read and write) a field.

In addition, you can define callers dynamically at runtime using the setup-*
functions setup-java-caller, setup-java-constructor and setup-
field-accessor, which are functions that match the define-* macros
above.

The setup-* functions effectively do exactly what the 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 terminol-
ogy.

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 my-probe-file which invokes the Java method
java.io.File.exists:

    (define-java-caller my-probe-file "java.io.File" "exists")

Define a Lisp function that calls one of the constructors of java.io.File:

    (define-java-constructor my-make-file "java.io.File")

At runtime, my-make-file will check which of the constructors of
java.io.File matches the arguments, and then call it.

See “Actual Java call” on page 173 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 sym-
bol 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 runtime, 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 runtime 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 "pack-age.class.method" to `call-java-method`. The actual runtime behavior is as described in “Actual Java call” on page 173.

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.

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 runtime behavior is as described in “Actual Java call” on page 173.
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 `7-0-0-0/etc/lispcalls.jar`, except on Android where it is part of the `7-0-0-0/`
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 inter-
face on the Java side, which is easier to switch between different implementa-
tions.

### 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. Other-
wise the pairs of V and A methods behave the same.
15.3 Calling from Java to Lisp

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 runtime 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:

   (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");
if (li instanceof View.OnTouchListener)   // allow for errors in Lisp
mTextView.setOnTouchListener((View.OnTouchListener)li);
```

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:

```lisp
(example-edit-file "android/android-othello-user")
```

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 runtime, 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
15.3 Calling from Java to Lisp

allows you to use runtime 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 \texttt{throw-an-exception} can be used to throw an exception from inside a call to a proxy function.

\section*{15.4 Working with Java arrays}

Java arrays are represented inside Lisp by a \texttt{jobject} or an instance of \texttt{standard-java-object}, like any other Java object. The function \texttt{java-array-element-type} returns the element type of a Java array or \texttt{nil} if it is not an array, and it is fast enough that it can be used as a predicate to determine whether a \texttt{jobject} represents an array.

\texttt{java-array-length} returns the length of a Java array.

\texttt{java-primitive-array-element-type} and \texttt{java-object-array-element-type} return the same values as \texttt{java-array-element-type} for an array of primitive type or an array of non-primitive type respectively, otherwise they return \texttt{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.

\subsection*{15.4.1 Accessing a single element}

The accessor \texttt{jvref} can be used to get and set (with \texttt{cl:setf}) the value in a Java "Vector" (that is, a one-dimensional array). For a multi-dimensional array, \texttt{jvref} gets and sets the first level "Vector", in other words it returns another array of one less dimension.

\texttt{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 \texttt{jvref} and \texttt{jaref} can be used to access arrays of any type. \texttt{jvref} is slightly faster, and does not allow passing wrong number of arguments.

\textbf{Note:} when accessing an element of a multi-dimensional array, \texttt{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
15.4 Working with Java arrays

(dotimes (z 10)
  (do-something (jaref java-array 3 4 z)))

use

(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:

(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, \texttt{map-java-object-array} accesses the top level elements, that is sub-arrays of one less dimension.

\texttt{primitive-array-to-lisp-array} and \texttt{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 \texttt{primitive-array-to-lisp-array} and \texttt{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>
<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>

\texttt{get-primitive-array-region} and \texttt{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.

\section*{15.5 Initialization of the Java interface}

The Java interface is a module which needs to be loaded by calling:

\begin{verbatim}
(requiring "java-interface")
\end{verbatim}
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 \texttt{init-java-interface}. \texttt{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 the system automatically calls \texttt{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 169.

\section*{15.6 Utilities and administration}

Use \texttt{jobject-p} to test whether a Lisp object is a \texttt{jobject} or not.

Use \texttt{lisp-java-instance-p} to test whether the argument is an instance of \texttt{standard-java-object}.

\texttt{get-jobject} returns the \texttt{jobject} for a Java object, \texttt{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 \texttt{standard-java-object}, \texttt{get-jobject} may return \texttt{nil} if the slot is not set. \texttt{ensure-is-jobject} is like \texttt{get-jobject}, but signals an error if its argument is not a \texttt{jobject}.

\texttt{jobject-class-name} can be used to find the Java class raw name of a Java object. \texttt{jobject-pretty-class-name} makes it "pretty", which matches how it appears in the Java code.

\texttt{jobject-string} returns a string representing the object the way Java wants to represent it (the result of \texttt{Object.toString}).

\texttt{jobject-to-lisp} and \texttt{lisp-to-jobject} can be used to convert between Lisp and Java objects of primitive types, which may sometimes be useful.

\texttt{find-java-class} can 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.

\texttt{jobject-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-constructo, 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 Java interface currently may generate at runtime 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 happen 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.
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`
Exception accessing a field (maybe wrong type of value).

`java-method-exception`
Exception inside a call to a Java method.

Serious exceptions:

`java-id-exception`
Failed to find JNI ID for a method.

`java-low-level-exception`
Failure in some JNI function.

### 15.7 Loading a LispWorks dynamic library into Java

When a LispWorks application is delivered as a dynamic library which is loaded by Java, the initialization function (first argument to `deliver`) should initialize the Java interface. But it is also possible to initialize the Java interface later, as long as the code does not try to use Java before doing the initialization.

It is not a good idea to export `JNI_OnLoad` (which is called automatically when Java loads a shared object), because it means the call to it will have to wait until LispWorks finishes initialization and is ready to accept callables, which will hang the caller until LispWorks finished initialization. On Unix you can add a piece of foreign code using the `deliver` keyword `:dll-added-files` which exports `JNI_OnLoad` that stores things only in C, maybe storing something that LispWorks can access later. It is probably a good idea to call
In this definition with timeout 0, so that LispWorks starts initializing in the background. Another option is to start another thread (using pthread_create) which calls into Lisp to do the initialization.

### 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 jobbject 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 jobbject 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 con- 
struct a jobbject 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 cre- 
ate 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 jobbject in an instance of standard-java-object can be read and writ- 
ten by the accessor java-instance-jobbject. Alternatively you can call create-instance-jobbject or create-instance-jobbject-list to create the 
jobbject for a given instance.

A simple interface for making an instance and its jobbject 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 stan- 
dard-java-object can be used to make the instance and the jobbject. Note, 
however, that the jobbject 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 jobbject 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-jobject-list with t on the result of make-instance.

If you have a jobject, and there is a CLOS class defined for its Java class, you can create a CLOS instance for it using create-instance-from-jobject. create-instance-from-jobject 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 (jobjects) 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 (jobjects) 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 job-
ject 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 Lisp instead. This avoids the creation of a jobject 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
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-7-0-0-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 initialized by the Java \texttt{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 \texttt{com.lispworks} package to support the Java/Lisp interface. This includes these classes:
  \begin{itemize}
  \item \texttt{com.lispworks.Manager}
    
    Defines the method \texttt{com.lispworks.Manager.init} to load and initialize LispWorks, error reporting interface and some basic utilities.
  \item \texttt{com.lispworks.LispCalls}
    
    Defines direct callers into Lisp, and support for Lisp proxies (which are Java proxies that call Lisp functions). \texttt{LispCalls} is really part of the general LispWorks Java Interface.
  \item \texttt{com.lispworks.BugFormLogsList} and \texttt{com.lispworks.BugFormViewer}
    
    Two activities to help display errors during development.
  \end{itemize}

- A few Android-specific interface functions:
  \begin{itemize}
  \item \texttt{android-funcall-in-main-thread}
  \item \texttt{android-funcall-in-main-thread-list}
  \item \texttt{android-get-current-activity}
  \item \texttt{android-main-thread-p}
  \end{itemize}

### 16.1 Delivering for Android

To use LispWorks in an Android project, the Android project needs three extra files:

1) \texttt{lispworks.jar}

This defines the support classes in the Java package \texttt{com.lispworks}. This file is part of the LispWorks distribution, and can be found in the \texttt{etc} directory in the LispWorks distribution:

\begin{verbatim}
(lispworks-file "etc/lispworks.jar")
\end{verbatim}
2,3) A LispWorks heap and dynamic library. These two files are generated by `deliver-to-android-project`. Both have the same base name (default "LispWorks"). The library has "lib" before the base name and ".so" after it. The heap file name is the library name with the string ".lwheap" appended. The heap file needs to be in the "assets" sub-directory of the project, while the dynamic library needs to be in `armeabi-v7a` sub-directory of the jni libs directory of the project. For Eclipse, the jni libs directory is `libs` at the top level, and `assets` is at the top too, so the files are by default in:

```
assets/LispWorks.so.lwheap
libs/armeabi-v7a/LispWorks.so
```

In Android Studio 0.4.6, the jni libs directory is `jniLibs`, and both it and the `assets` directory are in the "main" directory, so by default the files are in:

```
app/src/main/assets/LispWorks.so.lwheap
app/src/main/jniLibs/armeabi-v7a/LispWorks.so
```

By default, `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 `deliver-to-android-project`.

The `lispworks.jar` file is required so that your Java IDE knows about classes in `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 `com.lispworks.Manager.init` must be called to initialize LispWorks. If `library-name` was passed to `deliver-to-android-project`, then `com.lispworks.Manager.init` must be called with a matching name, otherwise the default "LispWorks" is used. `com.lispworks.Manager.init` can be called at any point during the lifetime of the Android app.

`com.lispworks.Manager.init` is asynchronous, in other words by the time it returns Lisp is not ready yet. `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 174 and “Using proxies” on page 175).

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-7-0-0-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 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. 

\( (\text{mp:ps}) \)

Shows the Lisp processes. Initially there are at least the idle process and the GUI process which displays as ”created by foreign code”.

2. 

\( (\text{setq } \ast\text{computer-plays-waste-time-in-seconds* } 2) \)

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 \( \ast\text{computer-plays-waste-time-in-seconds*} \) back to nil to make it behave normally.

3. 

\( (\text{defun eval-and-print (form)} \)
\( \text{(let ((res (eval form)))} \)
\( \text{(lw-ji:send-message-to-java-host} \)
\( \text{(princ-to-string res):reset})) \)

Defines a function to be used by the next two forms. Note that it uses \( \text{send-message-to-java-host} \) 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. 

\( (\text{eval-and-print } '(\text{mp:get-current-process})) \)
Use the function defined above to print the process in the current thread. That is the GUI process.

5.

(library:funcall-async 'eval-and-print '(library: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 (library:process-wait-for-event)))
       (lw-ji:format-to-java-host "~%got event ~s" event)
       (let ((res (library:general-handle-event event)))
         (lw-ji:format-to-java-host "~%Handling got ~s" res)))))
 (setq loop-executing-events-process
       (library: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.

(library:process-send loop-executing-events-process '(library: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>".

8.
16.3 The Othello demo for Android

(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)
   (dolimes (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"))
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.

```lisp
(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 *Lisp-Works 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 195).

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

```lisp
(example-edit-file "android/dialog")
```

and

```lisp
(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 two special images:

- `lispworks-7-0-0-x86-darwin-ios` which builds runtimes for the iOS Simulator running on Mac OS X, or
- `lispworks-7-0-0-arm-linux-ios` which builds runtimes for the real iOS device. To do this, you run the QEMU emulator on Mac OS X and tell it to run the `lispworks-7-0-0-arm-linux-ios` image.

There is an example script `examples/ios/run-lw-ios.sh` which can be used to invoke both 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, then you can conditionalize the filename passed to `deliver` to create two different files. For example,

```lisp
(deliver 'init-othello-server
(merge-pathnames
 (format nil "OthelloDemo/OthelloServer-~A.o"
    #+#arm "armv7"
    #+#x86 "i386")
   *project-path*)
  0
 :keep-symbols '(othello-server))
```

Then add both `OthelloServer-armv7.o` and `OthelloServer-i386.o` to the Xcode project. Building the project will result in a warning

```
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
```

(or vice versa for armv7 and i386). This warning 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 `LispWorksInitialize`. For example, `main` might be implemented like this:
17.3 Using Objective-C from Lisp

The function `objc:ensure-objc-initialized` must be called before you can use Objective-C from Lisp. A good place to call this is in the initialization function that is passed as the `func` argument to `deliver`.

For other details, see the *LispWorks Objective-C and Cocoa Interface User Guide and Reference Manual*.

17.4 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")

17.4.1 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.

**17.4.2 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 211).

The tiles in the board are dynamically positioned by the viewDidLayoutSubviews 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.
17.4 The Othello demo for iOS

17.4.3 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.

17.4.4 Notes about the Lisp code

The Lisp code triggered by the GUI is in the file

(example-edit-file "ios/ios-othello-user")

and uses the shared Othello logic in

(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`. 
18

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 214 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. 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

```
(funcallable-standard-object)
```

rather than

```
(funcallable-standard-object function)
```

so that LispWorks is compliant with the ANSI Common Lisp rules.

For details, see `funcallable-standard-object`, page 432.

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:
(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.
(in-package "CL-USER")

;; Metaclass of objects that might contain virtual slots.

(defclass 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)))

(defmethod slot-definition-allocation
  ((slotd virtual-slot-definition)) :virtual)

(defmethod (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)
(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 "~-% setf slot : ~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

object)
(call-next-method))

(defvar 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.

(defvar 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))

(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)`

```lisp
CL-USER 7 > (setf (real-slot object) 42)

setf slot : REAL-SLOT
42
```
19

Multiprocessing

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 and FreeBSD, LispWorks multiprocessing uses native threads and supports Symmetric Multi-processing (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 \texttt{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 235, 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 environment is already running. Note that, on Windows, Mac OS X, Linux, x86/x64 Solaris and FreeBSD, the LispWorks images shipped do start the programming environment. If you create an image which does not start the programming environment, 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

   ```lisp
   (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 163 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 228.

Note: On Windows, Linux, x86/x64 Solaris, FreeBSD and Mac OS X 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:

```lisp
(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.

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 231).

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 235 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 (mailboxes, barriers, semaphores and condition variables) is atomic. More information about these objects is in "Synchronization between threads" on page 240.

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 253).

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 235. 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 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 256).

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.
19.3.5 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 `set-array-single-thread-p`.

The result of parallel access to a "single-threaded" vector is unpredictable.

19.4 Locks

Locks can be used to control access to shared data by several processes.

The two main symbols used in locking are the function `make-lock`, to create a lock, and the macro `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:

```
<MP:LOCK "my-lock" Locked 2 times by "My Process" 2008CAD8>
```

The function `lock-owner` returns the process that locked a given lock.

The function `lock-name` returns the name of a lock.

The function `process-lock` blocks the current process until a given lock is claimed or a timeout passes, and `process-unlock` releases the lock.
The macro `with-lock` executes code with a lock held, and releases the lock on exit, as if by `process-lock` and `process-unlock`.

If you need to avoid blocking on a lock that is held by some other thread, then use `with-lock` with `timeout 0`, like this:

```lisp
(unless (mp:with-lock (lock nil 0)
  (code-to-run-if-locked)
  t)
  (code-to-run-if-not-locked))
```

The macros `with-sharing-lock` and `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.)
"Multiprocessing"

"Affinizing" 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`:

```
(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:

```
(mp:process-lock lock)
(unwind-protect
  (whatever)
  (mp:process-unlock lock))
```

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`:

```
(mp:process-lock lock)
(unwind-protect
  (whatever)
  (mp:process-unlock lock)
  (another-cleanup))
```

If `another-cleanup` is essential to execute in all throws, it needs its own `unwind-protect`:
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 inside a lock, or use 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: cl: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 \texttt{wait-reason} and \texttt{wait-function} arguments and potentially also arguments to pass to the \texttt{wait-function}. The \texttt{*-with-timeout} functions mentioned above also take a \texttt{timeout} argument. The \texttt{*-periodic-checks} functions also take a \texttt{period} argument.

The \texttt{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 \texttt{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.

\texttt{process-wait} and \texttt{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 earlier) 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 \texttt{process-poke}, if the waiting process is known, or by invoking the scheduler by \texttt{process-allow-scheduling}.

\textbf{Note:} All the specific Process Wait functions, described in “Specific Process Wait functions” on page 236, 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 \texttt{process-poke}, and should be be used in preference where possible.

A large disadvantage of \texttt{process-wait} and \texttt{process-wait-with-timeout} is that their \texttt{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 sys-
tem. 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 second 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 is 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 an `mp:mailbox` to pass the object (the `process-mailbox` of the receiver). It is possible to use a `mp: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. The receiver(s) use `mailbox-wait-for-event`, `mailbox-wait` or `mailbox-read`.
19.6 Process Waiting and communication between processes

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 untill 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 mp: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 variables or barriers. 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 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 241.

Barriers 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 `barrier-wait`). Barriers have additional features that allow more complex synchronization. For more details, see “Synchronization barriers” on page 242.

### 19.7 Synchronization between threads

In LispWorks 5.1 and previous versions, the main way to synchronize between threads is to use `mp:process-wait` or `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 `make-mailbox` and `mailbox-send`.
- Condition Variables (used with a lock). See “Condition variables” on page 241.
- Barriers (counting arrivals at a certain point in the code). See “Synchronization barriers” on page 242.
- Counting Semaphores (limiting the number of users of a shared resource). See “Counting semaphores” on page 243

Access to all of these objects is atomic and does not require additional locks (except for the lock that is already used with a condition variable).
19.7 Synchronization between threads

19.7.1 Condition variables

A 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 lock. The condition is typically something like data becoming available in a queue.

The function `condition-variable-wait` is used to wait for a condition variable to be signaled. It is always called with the lock held, which is automatically released while waiting and reclaimed before continuing. More than one thread can wait for a particular 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 `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.

19.7.2 Synchronization barriers

Barriers are objects 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:
(let ((barrier (mp:make-barrier num-of-processes)))
  (dotimes (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 the 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 number 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 \texttt{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 \texttt{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 \texttt{current-process-kill} is a convenient and safe way (as long as there are \texttt{unwind-protect} forms where needed) of causing processes to exit when they should.

\texttt{process-terminate} can be used to kill any process. If there is no Terminate Method (see \texttt{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 \texttt{unwind-protect} (unless they are executed with interrupts blocked) \texttt{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 \texttt{process-terminate} in actual applications.

19.8.2 Interrupting a process

\texttt{process-interrupt} and \texttt{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 \texttt{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 232) or low level atomic operations (see “Low level atomic operations” on page 253).

- **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 244 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*))
```

**New:** use low level atomic operations.

```lisp
(sys:atomic-push value *global-list*)
(sys:atomic-pop *global-list*)
```

### 19.8.4.3 Atomic push/delete

**Old:**

```lisp
(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.
19.8 Killing a process, interrupts and blocking interrupts

(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:

(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.
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`. 
19.10 Process properties

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

```
(mp:schedule-timer-relative
 (mp:make-timer 'capi:execute-with-interface
                'interface
                'capi:display-message "Time's up")
5)
```
or

```
(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`.


### 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.

#### 19.11.1.1 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 Other processes functions

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 \texttt{process-priority}. This can be changed using \texttt{change-process-priority}.

\begin{verbatim}
(mp:change-process-priority proc-1 10)
\end{verbatim}

Another way to specify the priority is to create the process with \texttt{process-run-function}, passing the keyword \texttt{:priority}:

\begin{verbatim}
(list
 (mp:process-run-function
  "SORTER-DOT" '(:priority 10) #'sorter \\.)
 (mp:process-run-function
  "SORTER-DASH" () #'sorter \\-))
\end{verbatim}

19.11.2 Accessing symbol values across processes

Use \texttt{symeval-in-process} to read the value of a dynamically bound symbol in a given process.

\begin{verbatim}
(setf mp:symeval-in-process)
\end{verbatim}

\texttt{symeval-in-process} can set the value of such a symbol. \texttt{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 \texttt{process-stop} and \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{process-unstop} on the worker.
4. The worker process proceeds and calls \texttt{process-stop}, and never wakes up.

To guard against this possibility, then the manager should call \texttt{process-stopped-p} when finding the worker in the second step above. Alternatively, it could check the result of \texttt{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 and FreeBSD

Each Lisp \texttt{mp:process} has a separate native thread and in LispWorks 6.0 and later versions these threads can run simultaneously.

\textbf{Note:} In LispWorks 5.1 and earlier versions, you can have many runnable \texttt{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 computer with multiple CPU cores. 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 \texttt{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 AIX/PowerPC and SPARC Solaris (not LispWorks for Linux, FreeBSD, x86/x64 Solaris or Macintosh).

### 19.12.3 Foreign callbacks on threads not created by Lisp

If a foreign callback occurs on a thread that was not created by LispWorks, then some data is kept on the Lisp side to make the foreign callback entry faster next time. This data is removed when the thread dies, but you force it to be removed sooner by calling `last-callback-on-thread`.

### 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>
<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>
</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.

Table 19.1 Places for low-level atomic operations

<table>
<thead>
<tr>
<th>Place</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A lexically bound symbol</td>
<td>It is an error to use a low level atomic operation on a lexically bound symbol.</td>
</tr>
<tr>
<td>(car cons)</td>
<td></td>
</tr>
<tr>
<td>(cdr cons)</td>
<td></td>
</tr>
<tr>
<td>(the type place)</td>
<td>For another place listed in this table.</td>
</tr>
<tr>
<td>(svref sv index)</td>
<td>Only simple-vector.</td>
</tr>
<tr>
<td>Structure accessors</td>
<td>The structure must be defined at compile time.</td>
</tr>
<tr>
<td>(slot-value object slot-name)</td>
<td>See below.</td>
</tr>
</tbody>
</table>
19.13.2 Aids for implementing modification checks

The macros \texttt{with-modification-check-macro} and \texttt{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 \texttt{with-modification-check-macro}, and the modifying code should be wrapped by the macro \texttt{with-modification-change}. They are associated by the fact that their \texttt{modification-place} argument is the same.

\texttt{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 \texttt{with-modification-change}.

\texttt{with-modification-check-macro} defines a lexical macro (by \texttt{macrolet}) with the name \texttt{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

(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-loads, ensure-memory-after-store, ensure-stores-after-memory and ensure-stores-after-stores.

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 inside 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:
(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 rea-
sons:

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:

(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:

(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.
20

Common Defsystem and ASDF

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 265 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.
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 969. 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 `:default-pathname` 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:

- The target(s). The action that is performed if the rule executes successfully.
20.2 Defining a system

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 \texttt{requires} clause is not already in the plan.

\section*{20.2.5 Examples}

Consider an example system, \texttt{demo}, defined as follows:

\begin{verbatim}
(defsystem demo (:package "USER")
  :members ("parent"
             "child1"
             "child2")
  :rules ((:in-order-to :compile ("child1" "child2")
          (:caused-by (:compile "parent"))
          (:requires (:load "parent"))))
\end{verbatim}

This system compiles and loads members into the \texttt{USER} package if the members themselves do not specify packages. The system contains three members --- \texttt{parent}, \texttt{child1}, and \texttt{child2} --- which may themselves be either files or other systems. There is only one explicit rule in the example. If \texttt{parent} needs to be compiled (for instance, if it has been changed), then this causes \texttt{child1} and \texttt{child2} to be compiled as well, irrespective of whether they themselves changed. In order for them to be compiled, \texttt{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 \texttt{parent}, then every member gets compiled. If the changed member is not \texttt{parent}, then \texttt{parent} must at least be loaded before compiling takes place.

The next example shows a system consisting of three files:

\begin{verbatim}
(defsystem my-system
  (:default-pathname "/junk/")
  :members ("a" "b" "c")
  :rules ((:in-order-to :compile ("c")
           (:caused-by (compile "b")))
          (:requires (load "a"))
          (:caused-by (compile "b")))))
\end{verbatim}

What plan is produced when all three files have already been compiled, but the file \texttt{b.lisp} has since been changed?
First, file `a.lisp` is considered. This file has already been compiled, so no instructions are added to the plan.

Second, file `b.lisp` is considered. Since this file has changed, the instruction `compile b` is added to the plan.

Finally file `c.lisp` is considered. Although this has already been compiled, the clause

\[ (:caused-by (:compile "b")) \]

causes the instruction `compile c` to be added to the plan. The compilation of `c.lisp` also requires that `a.lisp` is loaded, so the instruction `load a` is added to the plan first. This gives us the following plan:

1. Compile `b.lisp`.
2. Load `a.lisp`.
3. Compile `c.lisp`.

This last example shows how to make each fasl get loaded immediately after compiling it:

```
(defsystem my-system ()
  :members ("foo" "bar" "baz" "quux")
  :rules ((:in-order-to :compile :all
           (:requires (:load :previous)))))

(compile-system my-system :load t)
```

### 20.3 Using ASDF

You can load the supplied version of ASDF 2 by

```
(require "asdf")
```

Optionally, if you actually want your later version of ASDF 2, do

```
(asdf:load-system :asdf)
```

You may need to configure ASDF. For the language-level interface you should follow the ASDF documentation at [http://common-lisp.net/project/asdf/](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

(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

(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 runtime 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 two types, normal rules and error 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:

\[
\text{expression} \rightarrow \text{expression} \text{ operator} \text{ expression}
\]

with a semantic representation of a list of the individual semantic values, the Lisp grammar would contain the rule:

\[
((\text{expression} \text{ expression} \text{ operator} \text{ expression}) \ (\text{list} \ \$1 \ \$2 \ \$3))
\]

Error productions of the form:

\[
((\text{nt} \ :\text{error}) \ (\text{some error behavior}))
\]

are explained in the section below.

The first rule of the grammar should be of the form:

\[
((\text{nt} \ \text{nt1}) \ \$1)
\]

where the non-terminal $nt$ has no other productions and $nt1$ serves as the main “top-level” non-terminal.

### 21.2.2 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: \textit{statement} \rightarrow :if \ expression \ :then \ statement \ :else \ statement
- Rule b: \textit{statement} \rightarrow :if \ expression \ :then \ statement

this results in a conflict in the parser between a shift of \textbf{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 \textit{if} \ ... \ \textit{then} \ ... part of an \textit{if} \ ... \ \textit{then} \ ... \ \textit{else} \ ... statement being reduced, and a syntax error is produced for the \textbf{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 \textbf{(defparser \textit{name} \textit{grammar})} defines a number of functions. The main function \textit{name} is defined as the parsing function. For example:

\textbf{(defparser my-parser .. grammar .. )}

defines the function

\textit{my-parser lexer &optional symbol-to-string =>}

\textit{lexer} specifies the lexical analyzer function to be used. The optional argument \textit{symbol-to-string} should be a function mapping grammar symbols to strings for printing purposes. The default value of \textit{symbol-to-string} is the function \textbf{cl:identity}.

\textbf{defparser} also defines functions corresponding to the individual actions of the parser.

Normal actions are named:

\textit{name-action\textit{index}}

and error actions are named:

\textit{name-error-action\textit{index}}
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:

```
    non-terminal -> :error
```

This means that the parser accepts an erroneous string of tokens as constitut-
ing 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:
(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. Manipula-
tion 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")
(in-package "ENGLISH-PARSER")
(use-package '(parsergen))

;;; Define the parser itself.

(defun lex-english ()
  (let ((symbol (pop *input*)))
    (if symbol (get-lex-class symbol) nil)))

;;; Getting syntactic categories.

(defun lex-english ()
  (let ((symbol (pop *input*)))
    (if symbol (get-lex-class symbol) nil)))

;;; Define the parser itself.

(defun lex-english ()
  (let ((symbol (pop *input*)))
    (if symbol (get-lex-class symbol) nil)))

;;; Getting syntactic categories.

(defun lex-english ()
  (let ((symbol (pop *input*)))
    (if symbol (get-lex-class symbol) nil)))

;;; Getting syntactic categories.

(defun lex-english ()
  (let ((symbol (pop *input*)))
    (if symbol (get-lex-class symbol) nil)))
(sat :verb)(floor :noun))

(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 Dynamic Data Exchange

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 *conduct automatically managed conversations.
A function handling an automatically managed conversation takes a service designator and topic designator as two of its arguments, and either automatically establishes a conversation with a server responding to the service designator/topic designator pair, or uses an existing equivalent conversation. For the purpose of brevity, functions conducting automatically managed conversations are only briefly mentioned in this chapter. For the details see dde-advise-start*, dde-advise-stop*, dde-execute*, dde-execute-command*, 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 designator 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-client-advise-data. You can add methods to dde-client-advise-data specialized 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 particular 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
(defun my-conv (dde-client-conversation)
  ()
)```
22.2 Client interface

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 `dde-request` or `dde-request*`, the data contained in `item` is returned to the LispWorks client by the server.

In the case of a successful poke transaction with `dde-poke` or `dde-poke*`, the data provided is poked into `item` by the server.

The accessor `dde-item` (or `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.

### 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 277). LispWorks provides two ways of issuing an execute transaction, namely `dde-execute-string` and `dde-execute-command` (and the corresponding * functions that automatically manage conversations).

The following example shows how `dde-execute-string*` can issue a command to a server designated by :excel on the topic :system, in order to open a file called `foo.xls`:

```lisp
(dde-execute-string* :excel :system "[open("foo.xls")]")
```

The function `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 `dde-execute-command*` can issue the same command as in the previous example:

```lisp
(dde-execute-command* :excel :system 'open '("foo.xls"))
```
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 \texttt{define-dde-server}. Here the server class is called \texttt{foo-server} and it has the service name "\texttt{FOO}"
   \begin{verbatim}
   (define-dde-server foo-server "FOO")
   \end{verbatim}

2. Provide the server class with the functionality it requires by specializing methods on it and/or using \texttt{define-dde-server-function}. Here the server function is \texttt{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.
   \begin{verbatim}
   (define-dde-server-function (bar :topic :system)
      :execute
       ((x string))
       (format t "~&~s~%" x)
       t)
   \end{verbatim}

3. Start an instance of the server \texttt{foo-server} using \texttt{start-dde-server}.
   \begin{verbatim}
   (start-dde-server 'foo-server)
   \end{verbatim}
   This function returns the server object, which responds to requests for conversations with the service name "\texttt{FOO}", and accepts execute transactions for the function \texttt{bar} in the "\texttt{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 \texttt{dde-server-poke} and \texttt{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 sub-classes 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 \texttt{define-dde-server-function} by specifying a value of \texttt{:system} for the \texttt{topic} argument, as this would make the changes to the system topic visible to all users of DDE within the Lisp image.

Instead, specify \texttt{:server my-server :topic :system}, where \texttt{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.

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:
23 Common SQL

**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/re- lational 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>
</tr>
</thead>
<tbody>
<tr>
<td>Default database type</td>
<td>:odbc-driver</td>
<td>:oracle</td>
<td>:postgresql</td>
<td>:mysql</td>
</tr>
<tr>
<td>Other database type</td>
<td>:odbc</td>
<td>:oracle8</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>
</tr>
<tr>
<td>Mac OS X</td>
<td>MySQL Postgres</td>
<td>Oracle 10g and later (not supported on 64-bit PowerPC)</td>
<td>Postgres</td>
<td>MySQL</td>
</tr>
<tr>
<td>Linux</td>
<td>MySQL Postgres</td>
<td>Oracle 9i(r2) and later</td>
<td>Postgres</td>
<td>MySQL</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>None tested</td>
<td>Not supported</td>
<td>Postgres</td>
<td>MySQL</td>
</tr>
<tr>
<td>Solaris/Intel</td>
<td>None tested</td>
<td>Oracle 10g and later</td>
<td>Postgres</td>
<td>MySQL</td>
</tr>
<tr>
<td>AIX</td>
<td>None tested</td>
<td>Oracle 11 and later</td>
<td>None tested</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 runtime, 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" and "postgresql". However, not all platforms support all interfaces, see Table 23.1, page 287 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>
</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.

When a connection is no longer required, it should be closed by calling `disconnect`.
23.2 Initialization

The minimal code to initialize a connection is loading the code and connecting. For example, using only Oracle:

```
(receive "oracle")
(sql:connect "scott/tiger")
```

However, if you deliver an application then the `receive` call needs to happen at load time (before calling `deliver`), while the `connect` call must happen at runtime after the delivered application started. So your code should have two parts:

- In the script that loads the application code:
  ```
  (receive "oracle")
  ```
- In the code itself, at various places:
  ```
  (sql:connect "scott/tiger")
  ```

To get better error handling, you may want to add a call to `initialize-database-type`, in the startup function:

```
(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 288. 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

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:

dbname@:3307

hostname may also specify a unix socket name, which must start with the char-
acter '/'.

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

connection-spec can be a plist containing (some of) the keywords :username,

Each of these keywords may be omitted.

If :unix-socket is specified, then none of :hostname, :port and :connec-
tion 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
23.2 Initialization

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 \
-0 /usr/local/mysql/lib/libmysqlclient_r.dylib \n-all_load /usr/local/mysql/lib/libmysqlclient_r.a -lz
```

To build the 64-bit dynamic library:

```
gcc -m64 -dynamiclib -fno-common \n-0 /usr/local/mysql/lib/libmysqlclient_r.dylib \n-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

```
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.

23.3 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 310. For example, the following is a potential query and its result:

```lisp
(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

```lisp
(select [surname] :from [person] :flatp t)
=>
("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:
23.3 Functional interface

```
(select [*] :from [person])
=>
((2 111 "Brown") (3 222 "Jones") (4 333 "Smith"))
("ID" "Person_ID" "Surname")
```

```
(select [*] :from [person] :result-types '(:integer :string :string))
=>
((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:

```
(sql:select [Person_ID :string] [Surname] :from [person])
=>
(("111" "Brown") ("222" "Jones") ("333" "Smith"))
("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 325.

In this final example the :where keyword is used to specify a condition for returning selected values from the database.

```
=>
(("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:

```
(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:

```clojure
(insert-records :into [person]
  :attributes '(person_id income surname occupation)
  :values '(115 11000 "Johnson" "plumber"))
```

and:

```clojure
(insert-records :into [person]
  :av-pairs '((person_id 115)
              (income 11000)
              (surname "Johnson")
              (occupation "plumber")))
```

are equivalent to the following SQL:

```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:
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 `*cache-table-queries-default*`.

### 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`.

### 23.3.1.5 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
```
23.3  Functional interface

query-expression can be a string, a SQL expression (a result of the [...] syntax) or a prepared statement (a result of prepare-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.

```
(map-query nil '#(lambda (name) (print name))
   [select [ename] :from [emp] :flatp t])
```

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 "-&-20A-10D" name "N/A")
 finally
 (format t "-2&Av 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 (a result of the [..] syntax) or a prepared statement (a result of prepare-statement).

The function query runs a SQL query on a database and returns a list of values like select (see “Querying” on page 298). 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 1399.

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:

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 304.

2. Use sql-expression to construct the vendor-specific pieces of the SQL. The above expression can be written like this:
23.4 Object oriented interface

```
(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.

```
(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 special-
ized 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).

### 23.4.2 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 1383 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 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 (`eql`) 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)}
```
(select 'employee
    :where [= (slot-value 'employee 'employee-job) "SALESMAN"])
((#<db-instance EMPLOYEE 8067092>)
 (#<db-instance EMPLOYEE 8069536>)
 (#<db-instance EMPLOYEE 8069176>))
(list-classes)
(#<db-class EMPLOYEE> #<db-class DEPARTMENT>)

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 302): 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:

(loop for (jones company) being the tuples in
    [select 'person 'organization
     :where [= [slot-value 'person 'surname] "Jones"]]
    do (format t "~A ~A ~%" (slot-value jones 'forename) (slot-value company '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 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
23.5 Symbolic SQL syntax

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 317.

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 literal SQL

Each of these uses is demonstrated below.

23.5.1.1 Enclosing database identifiers

Database identifiers can be enclosed in the square bracket syntax as shown in the following examples.

```
[foo] => #<SQL-IDENT "FOO">
   This case corresponds to an unqualified SQL identifier as in: SELECT FOO FROM BAR.

[foo bar] => #<SQL-IDENT "FOO.BAR">
   This corresponds to a qualified SQL identifier as in:
   SELECT FOO.BAR FROM FOO

["foo" bar] => #<SQL-IDENT "\"foo\".BAR">
   This corresponds to a qualified SQL identifier with an aliased table name containing special characters as in:
   SELECT "foo".BAR FROM BAZ "foo".
```
23.5.1.2 SQL strings representing symbolic expressions

There are some SQL operators which may take a single argument (for example \texttt{any}, \texttt{some}, \texttt{all}, \texttt{not}, \texttt{union}, \texttt{intersect}, \texttt{except}, and \texttt{minus}). These are read as calls to the appropriate SQL operator. For example:

\begin{verbatim}
\[any \ ' (3 4)\] -> #<SQL-VALUE-EXP "(ANY (3,4))">
\end{verbatim}

This causes no conflict, however, as it is illegal to use these reserved words as identifiers in SQL. Similarly with two argument operators:

\begin{verbatim}
[> [baz] [beep]]
-> #<SQL-RELATIONAL-EXP "(BAZ > BEEP)".
\end{verbatim}

The \texttt{select} statement itself may be prepared for later query execution using the [{} syntax. For example:

\begin{verbatim}
[select [person_id] [surname] :from [person]]
\end{verbatim}

This form results in a SQL expression, which could be bound to a Lisp variable and later given to \texttt{query} to execute. For example:
23.5 Symbolic SQL syntax

```
[select [foo] [bar *]
 :from '([baz] [bar])
 :where [or [= [foo] 3]
       [> [baz.quux] 10]]
 ->
 #<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 315).
The general syntax is: `[<operator> <operand> ...], for instance:

(select [count [*]] :from [emp])

The operand can itself be a SQL expression, as in the following example:

(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 Symbolic SQL syntax

23.5.1.3 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:

```sql
(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-boolean-operator` and `sql-operator`.

23.5.1.4 Enclosing literal SQL

Literal SQL statements can simply be enclosed in the square bracket syntax, as shown below.

Creating a full query (which can be used as argument to `query`):

```sql
["SELECT FOO, BAR FROM BAZ"]
-> #<SQL "SELECT FOO, BAR FROM BAZ">
```

Using an unportable function condition in `:where`:

```sql
(sql:select [*] :from ["aTable"]
 :where ["unportable_function() > 89"])
```

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 1478.
The function `sql-expression` makes a SQL expression from the given keywords. This is equivalent to the first and third uses of the `{I}` syntax as discussed in Section 23.5.1 on page 311.

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 1473.

### 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
(let ((class (class-of object))
       (key-slots (db-class-keyfields class))
       (cols (map car (make-key-list object class)))
       (where `(=` (make-field-name class slot-name)
                  (lisp-to-sql-format
                   (slot-value object slot-name)
                   (if (listp slot-type)
                       (car slot-type)
                       slot-type)))
       (sql (apply (sql-operator 'and) cols)))
  (format t (sql-ppformat sql))
-> #<SQL-RELATIONAL-EXP "(EMP.EMPNO = 7369">"
```

Here is another example that produces a SQL `select` statement:
23.6 Working with date fields

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.
See also “Types of values returned from queries” on page 325 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
(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:

```sql
(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, 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`.

---

23 Common SQL
23.7 SQL I/O recording

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:

```sql
(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 1466 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`
  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 1475.

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 293 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 upercasing all names, so this may cause some mismatches. In general, Common SQL upercases 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 322 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 docu-
mented in connect, page 1366.

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:connect "me/mypassword/mydb"
                :sql-mode "ANSI_QUOTES")

See the description of the :sql-mode argument to connect, page 1366 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:

```lisp
(sql:map-query nil 'print
   "select forename, surname from people"
   :get-all t)

(sql:do-query
   ((forename surname) "select forename, surname from people"
    :get-all t)
   body)

(sql:simple-do-query
   (list "select forename, surname from people"
    :get-all t)
   body)

(loop for (forename surname) being each record
   "select forename, 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.

```lisp
(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 and :default-table-extra-options, and the :type and :extra-options keyword arguments to `create-table`.

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:

```sql
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 23.3 MySQL type mapping

<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>
<tr>
<td>Decimal</td>
<td>rational</td>
<td></td>
</tr>
<tr>
<td>All String types</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>All Binary types</td>
<td>(array (unsigned-byte 8) (*))</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>integer</td>
<td>Universal time</td>
</tr>
<tr>
<td>Datetime</td>
<td>integer</td>
<td>Universal time</td>
</tr>
<tr>
<td>Timestamp</td>
<td>integer</td>
<td>Universal time</td>
</tr>
<tr>
<td>Time</td>
<td>integer</td>
<td>Number of seconds</td>
</tr>
<tr>
<td>Year</td>
<td>integer</td>
<td>Number of years</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 298.
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.

23.9.10 Autocommit
Common SQL sets autocommit to 0 when it opens a MySQL connection.

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 327.

23.10.1 Connection specification
See “Connecting to Oracle” on page 292 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 1MB 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 328 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:

```
(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 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 1466 for details. For example, writing a line in the end of the log file of station number 573:

```lisp
create table logfiles (stationid integer, logfiles clob)
.. insert records ..

(sql:do-query ((log-stream)
  (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:

```lisp
(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)

(s 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 Oracle LOB interface

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`

```c
int my_lob_processor(OCIHandle *lob,
                     OCISvcCtx *Context,
                     int other_arg)
```

(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:

```c
(my-lob-processor (sql:ora-lob-lob loc)
                   (sql:ora-lob-svc-ctx-handle loc)
                   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><code>ora-lob-lob-locator</code></td>
<td><code>p-oci-lob-locator</code></td>
<td><code>OCIlobLocator*</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>p-oci-file</code></td>
<td></td>
</tr>
<tr>
<td>context</td>
<td><code>ora-lob-svc-ctx-handle</code></td>
<td><code>p-oci-svc-ctx</code></td>
<td><code>OCISvcCtx*</code></td>
</tr>
<tr>
<td>environment</td>
<td><code>ora-lob-env-handle</code></td>
<td><code>p-oci-env</code></td>
<td><code>OCIEnv*</code></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 330. 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 601. 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 333.

`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 331. 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 329.

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 `lob-stream` interface may be more efficient.

Note also that the difference in efficiency between the direct calls and the `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 `ora-lob-open` and `ora-lob-close`.

You can read data from the LOB locator into a Lisp buffer or foreign buffer using `ora-lob-read-buffer` and `ora-lob-read-foreign-buffer` respectively.

Similarly `ora-lob-write-buffer` and `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 `ora-lob-get-buffer`.

`ora-lob-read-into-plain-file` writes the contents of a LOB into a file.

`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 `ora-lob-create-temporary`.

You can test whether a LOB is temporary with `ora-lob-is-temporary`.

You can free a temporary LOB locator if necessary with `ora-lob-free-temporary`, though temporary LOB locators are freed automatically when the database connection is closed by `disconnect`.

### 23.11.9.7 Control of buffering

These functions control the internal buffering by the Oracle client: `ora-lob-enable-buffering`, `ora-lob-disable-buffering`, and `ora-lob-flush-buffer`. They have no interaction with any of the other functions above.
23.12  Using ODBC

23.12.1  Configuring unixODBC
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:

(requires "odbc")

At runtime, Common SQL automatically loads the unixODBC module from the location in the variable sql:*odbc-foreign-modules*. In LispWorks for Linux this variable initially has the value ("/usr/lib/libodbc.so"). Therefore if, for example, the runtime machine unixODBC installed in /usr/local/, at runtime do:

(setq sql:*odbc-foreign-modules* '("/usr/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*.
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:

```
(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.

```
(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
24.2 An illustrative example of user defined streams

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.

```
(defmethod input-stream-p ((stream unicode-ls-stream))
  (input-stream-p (ls-stream-file-stream stream)))
(defmethod output-stream-p ((stream unicode-ls-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.

```
(defun stream:stream-read-char ((stream unicode-ls-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.
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)))

(defmethod stream:stream-clear-input ((stream unicode-ls-stream))
  (clear-input (ls-stream-file-stream stream)))
\end{verbatim}

### 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 \#\texttt{Newline}, then the method writes a \#\texttt{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.
24.2  An illustrative example of user defined streams

(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 argument.

(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.
24.2 An illustrative example of user defined streams

(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))))
User Defined Streams
TCP and UDP socket communication and SSL

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 451.

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/7-0-0-0/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 num-
bers 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 connec-
tions. 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-host-entry` 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 Socket Stream SSL interface

The Socket Stream SSL interface allows you to use Secure Socket Layer (SSL) with Lisp objects of type socket-stream.

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.

The Socket Stream SSL interface is in the "comm" module, so to load it you evaluate

(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.

25.7.1 Creating a stream with SSL

There are three ways to make a socket-stream with SSL processing:

- Call (make-instance 'socket-stream :ssl-ctx ...)
- Call (open-tcp-stream ... :ssl-ctx ...)
- Call attach-ssl on a socket-stream.

For example:

(open-tcp-stream some-url 443 :ssl-ctx t)

25.7.2 SSL-CTX and SSL objects

When the value of the :ssl-ctx argument is a symbol, LispWorks automatically creates an SSL_CTX object and an SSL object and uses them. If you need to configure these objects, you can access them by the following methods:

- When passing :ssl-ctx or when calling attach-ssl (as described above) also pass :ctx-configure-callback and :ssl-configure-callback.
- Use the accessors socket-stream-ssl and socket-stream-ctx.
- Make your own SSL-CTX or SSL objects and pass them as the ssl-ctx argument.
25.7.3 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`.

25.7.3.1 OpenSSL constants

The Lisp constants `SSL_FILETYPE_ASN1` and `SSL_FILETYPE_PEM` representing file types are provided.

25.7.3.2 Naming conventions for direct OpenSSL calls

This section describes the mapping between OpenSSL function names and the corresponding Lisp names.

25.7.3.3 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.

25.7.3.4 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"

25.7.4 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 25.1 Direct calls to OpenSSL

<table>
<thead>
<tr>
<th>Lisp function</th>
<th>Return values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssl-add-client-ca</td>
<td></td>
</tr>
<tr>
<td>ssl-cipher-get-bits</td>
<td>First value is number of bits the cipher actually uses. Second value is number of bits the algorithm of the cipher can use (which may be higher).</td>
</tr>
<tr>
<td>ssl-cipher-get-name</td>
<td>string. e.g. &quot;DHE-RSA-AES256-SHA&quot;</td>
</tr>
<tr>
<td>ssl-cipher-get-version</td>
<td>string. e.g. &quot;TLSv1/SSLv3&quot;</td>
</tr>
<tr>
<td>ssl-clear-num-renegotiations</td>
<td></td>
</tr>
<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>
</tbody>
</table>
Table 25.1 Direct calls to OpenSSL

<table>
<thead>
<tr>
<th>Lisp function</th>
<th>Return values</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>ssl-ctx-use-certificate-file</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-use-privatekey-file</td>
<td></td>
</tr>
<tr>
<td>ssl-ctx-use-rsaprivatekey-file</td>
<td></td>
</tr>
<tr>
<td>ssl-get-current-cipher</td>
<td>ssl-cipher-pointer</td>
</tr>
<tr>
<td></td>
<td>Can be a null pointer.</td>
</tr>
<tr>
<td>ssl-get-max-cert-list</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.

Table 25.1 Direct calls to OpenSSL

<table>
<thead>
<tr>
<th>Lisp function</th>
<th>Return values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssl-get-mode</td>
<td></td>
</tr>
<tr>
<td>ssl-get-options</td>
<td></td>
</tr>
<tr>
<td>ssl-get-verify-mode</td>
<td>integer</td>
</tr>
<tr>
<td>ssl-get-version</td>
<td>string</td>
</tr>
<tr>
<td>&quot;TLSv1&quot;, &quot;SSLv2&quot; or &quot;SSLv3&quot;</td>
<td></td>
</tr>
<tr>
<td>ssl-load-client-ca-file</td>
<td></td>
</tr>
<tr>
<td>ssl-need-tmp-rsa</td>
<td></td>
</tr>
<tr>
<td>ssl-num-renegotiations</td>
<td></td>
</tr>
<tr>
<td>ssl-session-reused</td>
<td>None</td>
</tr>
<tr>
<td>ssl-set-accept-state</td>
<td>None</td>
</tr>
<tr>
<td>ssl-set-client-ca-list</td>
<td></td>
</tr>
<tr>
<td>ssl-set-connect-state</td>
<td>None</td>
</tr>
<tr>
<td>ssl-set-max-cert-list</td>
<td></td>
</tr>
<tr>
<td>ssl-set-mode</td>
<td></td>
</tr>
<tr>
<td>ssl-set-options</td>
<td></td>
</tr>
<tr>
<td>ssl-set-tmp-rsa</td>
<td></td>
</tr>
<tr>
<td>ssl-set-tmp-dh</td>
<td></td>
</tr>
<tr>
<td>ssl-total-renegotiations</td>
<td></td>
</tr>
<tr>
<td>ssl-use-certificate-file</td>
<td></td>
</tr>
<tr>
<td>ssl-use-rsaprikeye-key-file</td>
<td></td>
</tr>
<tr>
<td>ssl-use-privatekey-file</td>
<td></td>
</tr>
</tbody>
</table>
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.

### 25.7.5 Socket Stream SSL keyword arguments

The keyword arguments `:ssl-ctx`, `:ssl-side`, `:ctx-configure-callback` and `:ssl-configure-callback` can be passed to create and configure socket streams with SSL processing. The various methods for creating and configuring SSL streams accept these keyword arguments as shown in Table 25.2, page 356.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>make-instance</code></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>open-tcp-stream</code></td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>attach-ssl</code></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>make-ssl-ctx</code></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</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.

`: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
25.7 Socket Stream SSL interface

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 is closed. make-instance, attach-ssl and open-tcp-stream also make an SSL object, use it and free it when the stream 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. make-instance, attach-ssl and open-tcp-stream 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 make-instance, attach-ssl and open-tcp-stream. This maybe 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-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.

:ssl-side specifies which side the stream is. The value ssl-side can be one of :client, :server or :both. open-tcp-stream does not take this keyword and always uses :client. For the other calls this argument defaults to :server. The value of ssl-side is used in two cases:

When a new SSL_CTX object is created, it is used to select the method:

:client => *_client_method
When a new SSL object is created, when \texttt{ssl-side} is either \texttt{:client} or \texttt{:server}, LispWorks calls \texttt{ssl-set-connect-state} or \texttt{ssl-set-accept-state} respectively.

If the value of \texttt{ssl-ctx} is a \texttt{ssl-pointer}, \texttt{ssl-side} is ignored.

\texttt{:ctx-configure-callback} specifies a callback, a function which takes a foreign pointer of type \texttt{ssl-ctx-pointer}. This is called immediately after a new SSL\_CTX is created. If the value of \texttt{ssl-ctx} is not a symbol, \texttt{ctx-configure-callback} is ignored.

\texttt{:ssl-configure-callback} specifies a callback, a function which takes a foreign pointer of type \texttt{ssl-pointer}. This is called immediately after a new SSL is created. If the value of \texttt{ssl-ctx} is not a \texttt{ssl-pointer}, \texttt{ssl-configure-callback} is ignored.

### 25.7.6 Attaching SSL to an existing socket-stream

You can attach SSL to an existing socket-stream by calling \texttt{attach-ssl} on the stream. \texttt{attach-ssl} ensures the OpenSSL library is loaded and seeds the Pseudo Random Number Generator (PRNG). The socket-stream SSL keyword arguments are processed by \texttt{attach-ssl} as described in “Socket Stream SSL keyword arguments” on page 356.

Detach SSL from a socket-stream and shut down the SSL with \texttt{detach-ssl}.

For full descriptions see \texttt{attach-ssl}, page 480 and \texttt{detach-ssl}, page 498.

### 25.7.7 Using SSL objects directly

The C objects SSL and SSL\_CTX are represented in LispWorks by foreign pointers with type \texttt{ssl-pointer} and \texttt{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 \texttt{: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
25.7 Socket Stream SSL interface

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:

```lisp
(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))
  (ssl-ctx-free 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.

25.7.8 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 initialize 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.
25.7.9 Obtaining and installing the OpenSSL library

At the time of writing, OpenSSL is available as shown in Table 25.3:

<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>
<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</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 sunfreeware.com for both 32-bit and 64-bit.</td>
</tr>
</tbody>
</table>

25.7.9.1 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.

25.7.9.2 Loading 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 25.4, page 361.
Asynchronous I/O

On machines where the path is unknown or is incorrect, you must set the path. Do this by calling `set-ssl-library-path`, or by passing the path as the `library-path` argument to `ensure-ssl`.

25.7.10 Errors in SSL

If there are errors inside SSL, LispWorks will signal an error of type `ssl-condition`, 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.

25.7.11 Examples of using the socket stream SSL interface

See the example files in:

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

25.8 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 func-

---

**Table 25.4 Loading the OpenSSL libraries**

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td><code>-lssl</code></td>
</tr>
<tr>
<td>Windows</td>
<td><code>libeay32.dll</code></td>
</tr>
<tr>
<td></td>
<td><code>libssl32.dll</code></td>
</tr>
<tr>
<td>Solaris</td>
<td><code>-lssl</code></td>
</tr>
<tr>
<td>AIX</td>
<td><code>-lssl</code></td>
</tr>
<tr>
<td>Mac OS X</td>
<td><code>-lssl</code></td>
</tr>
<tr>
<td>Others</td>
<td><code>nil</code></td>
</tr>
</tbody>
</table>
tion that returns a value (like `cl:read-line`). 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.

### 25.8.1 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 363 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 365).

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 363 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")

### 25.8.2 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.

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

```lisp
(ex example-edit-file "async-io/multiplication-table")
(ex example-edit-file "async-io/print-connection-delay")
```

### 25.8.3 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 366). 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 `async-io-state` function.
Another possible solution is to perform operations that can be fast using one `wait-state-collection`, and perform slow operations on (an)other `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" `wait-state-collection`, but communicate with the accepted connection on a slow `wait-state-collection` (pass `:create-state nil` to `accept-tcp-connections-creating-async-io-states`, and in the callback use `create-async-io-state` with another `wait-state-collection`). You may also decide to do the communication using streams and synchronous I/O (pass `:create-state nil` and in the callback use `(make-instance 'socket-stream ...)`) and send the result to another process.

### 25.8.4 Asynchronous I/O and multiprocessing

Processing of the `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:

- `loop-processing-wait-state-collection`
- `call-wait-state-collection`
- `wait-for-wait-state-collection`
- `close-wait-state-collection`

`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 `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 (`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`. 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:

```lisp
(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:
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 (\texttt{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 \texttt{(make-instance 'comm:socket-stream ...)} to do the actual communication.

When using Java sockets, the SSL configuration arguments \texttt{ctx-configure-callback} and \texttt{ssl-configure-callback}, as well as the \texttt{write-timeout} and \texttt{ipv6}, are ignored. The \texttt{ssl-ctx} is ignored when passed to \texttt{cl:make-instance}, and when passed to \texttt{open-tcp-stream} or \texttt{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 \texttt{javax.net.SocketFactory}) to \texttt{open-tcp-stream-using-java}. The application needs to set up and configure this factory using Java methods. By default, \texttt{open-tcp-stream} and \texttt{open-tcp-stream-using-java} use the default factory (which they get by the method \texttt{getDefault} on \texttt{javax.net.SocketFactory} or \texttt{javax.net.ssl.SSLSocket}). Thus configuring the default factories affects what they do.

\texttt{cl:listen} is unreliable because the only way to check whether there is input on a Java socket is to use the Java method \texttt{"available"} on the input stream of the Java socket (that is, the result of the method \texttt{"getInputStream"}). The \texttt{"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 \texttt{"available"} on an input stream of a socket is reliable, then you can trust \texttt{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 \texttt{init-java-interface}. To load the LispWorks Java interface, do

\begin{verbatim}
(require "java-interface")
\end{verbatim}

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 \texttt{httpclient} from \texttt{apache.org}, so \texttt{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 cl:listen, discussed above. In principle, if you can find OpenSSL library for Android you can switch it back by calling switch-open-tcp-stream-with-ssl-to-java with nil, and use SSL in the usual way. You need to use 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 java-exception error with exception NetworkOnMainThreadException. That applies to 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.
TCP and UDP socket communication and SSL
26

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 - #xffff) 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, #xdfff]. 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 Character and String types

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` is 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+00a0` 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 374).

Compatibility note: **bmp-string** is 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 379. For example:
26.3 Character and String types

CL-USER 1 > (set-default-character-element-type 'base-char)
BASE-CHAR

CL-USER 2 > (coerce (list #
Ideographic-Space) 'string)
Error: #
Ideographic-Space is not of type BASE-CHAR.
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 3 : 1 > :a

CL-USER 4 > (set-default-character-element-type 'character)
CHARACTER

CL-USER 5 > (coerce (list #
Ideographic-Space) 'string)
" "

The following types are subtypes of cl: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-chars.

simple-base-string
hold any base-char.

simple-bmp-string
hold any bmp-char.

simple-text-string
hold 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 379.

26.3.5.1 String types at runtime

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)
ambiguously 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

```lisp
(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:

(make-string 3)

returns a simple-base-string and

(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

(make-string 3)

returns a simple-text-string and

(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:

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*.

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

```lisp
(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 with default native byte order. See “16-bit External formats guide” on page 383 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 383 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 383 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/7-0-0-0/etc/. It is also used at runtime 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

Windows code page 936. The encoding data is read from a file windows-936-2000.ucm and is pre-built into LispWorks.
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 #xdbff) 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 use 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 :latin1.

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 #xdbff).

: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 Unix/Linux/FreeBSD/Mac OS X 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:

```lisp
(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 \texttt{open} or \texttt{with-open-file} gets a non-complete \texttt{:external-format} argument \texttt{ef-spec} then the system decides which external format to use by calling the function \texttt{guess-external-format}.

The default behavior of \texttt{guess-external-format} is as follows:

1. When \texttt{ef-spec}'s name is \texttt{:default}, this finds a match based on the file-name; 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 \texttt{ef-spec}'s name is assumed to name an encoding and this encoding is used.

2. When \texttt{ef-spec} does not include the \texttt{:eol-style} parameter, it then also analyses 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:

\begin{verbatim}
(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)
\end{verbatim}

The behavior of \texttt{guess-external-format} is configurable via the variables \texttt{*file-encoding-detection-algorithm*} and \texttt{*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 \texttt{open}, \texttt{compile-file} and so on, you can modify the value of \texttt{*file-encoding-detection-algorithm*}.

For example given the following definition:
(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:

(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:

(setq system:*file-encoding-detection-algorithm*
  '(utf-8-file-encoding))

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 super-type of the type of characters produced by the external format.
2. If direction is :output or :io, the element-type argument must be a sub-type of the type of characters accepted by the external format

If the element-type argument does not satisfy these requirements, an error is signaled.

If element-type is :default the system chooses the stream-element-type on the basis of the external format.

26.8.5 External formats and the LispWorks Editor

The LispWorks Editor uses open with :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 *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 383 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 *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 `cl:char-equal`, `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 `cl:alpha-char-p`, `cl:both-case-p` and so on.

#### 26.10.1 Unicode case insensitive character comparison

The functions `unicode-char-equal`, `unicode-char-not-equal`, `unicode-char-lessp`, `unicode-char-not-lessp`, `unicode-char-greaterp` and `unicode-char-not-greaterp` compare characters similarly to `cl:char-equal` etc, but using Unicode’s simple case folding rules.
26.10.2 Unicode case insensitive string comparison

The functions `unicode-string-equal`, `unicode-string-not-equal`, `unicode-string-lessp`, `unicode-string-not-lessp`, `unicode-string-greaterp` and `unicode-string-not-greaterp` compare strings similarly to `cl:string-equal` etc, but using Unicode's simple case folding rules.

26.10.3 Unicode character predicates

The predicates `unicode-alphanumeric-p`, `unicode-alpha-char-p`, `unicode-lower-case-p`, `unicode-upper-case-p` and `unicode-both-case-p` test for properties of a character in Unicode's "general category".
Internationalization: characters, strings and encodings
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` can be used to distinguish between systems based on Windows 95 and those based on Windows NT. `software-version` allows you to identify variants such as Windows Vista, Windows 7, Windows 8 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`:

```lisp
(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-7-0-0-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 150 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.

-env A synonym for -environment.

-display display Sets the X display to use when starting a LispWorks GUI on X Windows.

-IOPhost host Controls the host name in placed in IORs. See Developing Component Software with CORBA for details.

-IOPnumeric
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 146.

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 150 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 868.

`-ORBport orbport`
27.4 The Command Line

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 396. 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 7.0.

--reserve-size ReserveSize
Specifies the reserve size on supported platforms, as described in “Startup relocation” on page 396.

-siteinit siteinit-file
siteinit-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:

(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 396.

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 #x10000000. In principle it can grow to almost #x80000000 (the libraries are at higher addresses).

For the other platforms and for 64-bit LispWorks, see the discussion in “Startup relocation” on page 396.

27.5.3 Reporting current allocation

The simplest way to see the current Lisp allocation is to call (room t).

To obtain values representing the current total allocation, call 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 and FreeBSD. The 32-bit LispWorks implementations on non-x86 platforms are not relocatable. 64-bit LispWorks is relocatable on all sup-
27.6 Startup relocation

ported 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, and LispWorks (64-bit) for SPARC/Solaris - 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 on non-Windows platforms, pass one or both of these command line arguments:

\[ --\text{relocate-image} \ BaseAddress \]

The base address, interpreted as a hexadecimal number by calling `strtol(BaseAddress, NULL, 16)`

\[ --\text{reserve-size} \ ReserveSize \]

The reserve size, interpreted as a hexadecimal number by calling `strtol(ReserveSize, NULL, 16)`

There is currently no way to control the relocation of a LispWorks for Windows executable.

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 122.

### 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** (256GB).

### 27.6.3.1 Linux

On old Linux systems LispWorks (64-bit) for Linux has range 192GB, 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 150GB to 160GB. 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 768GB, ending at **#x10000000000**. 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 **#x3c000000000** (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”.

On Mac OS X an Objective-C API is provided. See the *LispWorks Objective-C and Cocoa Interface User Guide and Reference Manual*.

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 868. 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 149 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 406.)

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 runtime with `user-preference` and `(setf user-preference)`.

#### 27.12.3 Example using user preferences

Define a registry path:

```
(setf (sys:product-registry-path :deep-thought)  
      '("Software" "My Company" "Deep Thought")
```
Store a preference for the current user:

```
(setf (user-preference "Answers"
    "Ultimate Question"
    :product :deep-thought)
  42)
```

Retrieve a preference for the current user, potentially in a subsequent session:

```
(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 8, Windows 7 and Windows Vista

C:\ProgramData

Windows XP (now unsupported)

C:\Documents and Settings\All Users.WINDOWS\Application Data

Mac OS X

/Library/Application Support
27.14 Special locations in the file system

Android

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 runtime. 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:

```
(sys:get-folder-path :common-appdata)
```

Here is another example of differences between operating systems. On Windows 7 and Windows Vista:

```
(sys:get-folder-path :my-documents)
=> #P"C:/Users/dubya/Documents/"
```

On Windows 98 SE:

```
(sys:get-folder-path :my-documents)
=> #P"C:/My Documents/"
```

On Mac OS X:

```
(sys:get-folder-path :my-documents)
=> #P"/u/ldisk/dubya/Documents/"
```

See `get-folder-path` for more details.

On Windows NT-based systems 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:

```lisp
(create-temp-file :prefix "LW")
=> #P:\DOCUME~1/dubya/LOCALS~1/Temp/LW383vwfZtN.tmp"
```

On Linux:

```lisp
(create-temp-file :prefix "LW")
=> #P"/tmp/LWladokNa.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))
        (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 filesystems are case-sensitive).
LispWorks’ Operating Environment
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.


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

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

\[(\text{optimize (float 0)})\]

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:

\[\text{(defun incf-signed-byte-64 (ptr index)}\]
\[\text{(declare (optimize (safety 0) (float 0)))}\]
\[\text{(type fixnum index))}\]
\[\text{(setf (fli:foreign-typed-aref 'sys:int64 ptr index)}\]
\[\text{(sys:int64-1+ (fli:foreign-typed-aref 'sys:int64 ptr index))}\]
\[\text{;; return ptr, since otherwise the int64 would}\]
\[\text{;; need to be boxed to return it}\]
\[\text{ptr)}\]

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 16TB 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 (3.75TB)</td>
<td></td>
</tr>
<tr>
<td>PowerPC Macintosh</td>
<td>#x40000000000 to #x40000000000 (3.75TB)</td>
<td></td>
</tr>
<tr>
<td>old Linux</td>
<td>#x40000000000 to #x70000000000 (192GB)</td>
<td>Effective limit around 160GB.</td>
</tr>
<tr>
<td>modern Linux</td>
<td>#x40000000000 to #x44000000000 (4TB)</td>
<td></td>
</tr>
<tr>
<td>Windows</td>
<td>#x40000000000 to #x40000000000 (3.75TB)</td>
<td></td>
</tr>
<tr>
<td>Solaris</td>
<td>#x40000000000 to #x10000000000 (768GB)</td>
<td></td>
</tr>
</tbody>
</table>

In contrast, 32-bit LispWorks has a maximum heap size of 1.5-3.0GB 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 #x7000000000. On new systems, it sets the end to #x44000000000, thus giving a range of 4TB. However, if the size is given explicitly by command line argument --reserve-size or 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 398 and “Startup relocation of 64-bit LispWorks” on page 398.
29.3 Architectural constants

Common Lisp constants have the values shown in Table 29.2

Table 29.2 Architectural constants

<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 117 and “Memory Management in 64-bit LispWorks” on page 128.

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 “Socket Stream SSL interface” on page 351:
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 361:

(example-edit-file "async-io/driver")
(example-edit-file "async-io/multiplication-table")
(example-edit-file "async-io/print-connection-delay")

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 275:

(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
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 147 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 265:

(example-edit-file "misc/asdf-integration")
30 Self-contained examples
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`.  

423
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.

```lisp
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`
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
Signature: `compute-effective-method-function-from-classes gf classes => em-function`

Arguments:
- `gf`: A generic function.
- `classes`: A list of class metaobjects.

Values: `em-function` is a function or `nil`.

Description:
The generic function `compute-effective-method-function-from-classes` 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 `no-applicable-method` is called. Otherwise, `em-function` may be cached by the generic function and is called with the arguments supplied to the generic function.

The default method for `compute-effective-method-function-from-classes` 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 `compute-effective-method-function-from-classes` 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.

Example:
A “computed” 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.
(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)))

(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 callable-standard-object.

Values
target
A standard-object, but not a callable-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 callable-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 stdandard-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
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."))
  `(,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.
The generic function process-a-slot-option describes how the value of a slot option is parsed. It is called at 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.

processed-options should be a plist of slot initargs and values containing those from already-processed-other-options together with initargs for option as required. These are added to any other initargs for the slot.

Example

(defun extended-class (standard-class) ()
  
  (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)))

(defun extended-direct-slot-definition
clos:standard-direct-slot-definition
  ((extended-slot :initarg :extended-slot :initform
                  nil)))

(defun clos:direct-slot-definition-class
  ((x extended-class) &rest initargs)
     'extended-direct-slot-definition)

(defun test ()
  ((:regular :initform 3)
   (:extended :extended-slot t :initform 4))

To add a slot option :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 :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))
     (list* :special-reader
         `''(,value)
         already-processed-options)
     (call-next-method))))

See also defclass process-a-class-option

replace-standard-object

Function

Summary
Replaces the values in a CLOS object’s slots by the values of slots from another object.

Package clos

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

```
(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.

2. `set-clos-initarg-checking` supersedes

#### See also
- `class-extra-initargs`
- `compute-class-potential-initargs`
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  \textit{slot-boundp-using-class class object slot-name => result}

Arguments  
- \textit{class}  A class metaobject, the class of \textit{object}.
- \textit{object} An object.
- \textit{slot-name} A slot name.

Values  
- \textit{result} A boolean.

Description  
The generic function \textit{slot-boundp-using-class} implements the behavior of the \textit{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 \textit{t} for this argument.

See also  
- \textit{slot-makunbound-using-class}
- \textit{slot-value-using-class}

\textbf{slot-makunbound-using-class} \hspace{1cm} \textit{Generic Function}

Summary  
Implements \textit{slot-makunbound}.

Package  
clos

Signature  \textit{slot-makunbound-using-class class object slot-name => object}

Arguments  
- \textit{class}  A class metaobject, the class of \textit{object}.
- \textit{object} An object.
- \textit{slot-name} A slot name.

Values  
- \textit{object} The \textit{object} argument.

Description  
The generic function \textit{slot-makunbound-using-class} implements the behavior of the \textit{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.
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

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace-on-access</td>
</tr>
<tr>
<td>Summary</td>
</tr>
<tr>
<td>Package</td>
</tr>
<tr>
<td>Signature</td>
</tr>
</tbody>
</table>

| 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.

<table>
<thead>
<tr>
<th>Values</th>
<th>Returns t.</th>
</tr>
</thead>
</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.
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.

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

```lisp
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 => accepting-handle
```

### Arguments

- **collection**
  - A `wait-state-collection`
- **service**
  - An integer, a string or `nil`
- **connection-function**
  - A function designator.
- **init-function**
  - `nil` or a function designator.
- **init-timeout**
  - `nil` or a non-negative real number.
- **backlog**
  - `nil` or a positive integer.
- **address**
  - An integer, an `ipv6-address` object, a string or `nil`.
- **nodelay**
  - A generalized boolean.
- **keepalive**
  - A generalized boolean.
- **ipv6**
  - The keyword `:any`, `nil`, `t` or the keyword `:both`.
- **create-state**
  - A boolean.
- **queue-output**
  - A boolean.
- **name**
  - A Lisp object.
- **handle-name**
  - A Lisp object.
- **user-info**
  - A Lisp object.

### 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 348.
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.

For details of backlog, address, nodelay, keepalive and ipv6, see start-up-server.

The default value of nodelay is \texttt{t}.

The default value of ipv6 is \texttt{: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 363
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

Signature

accepting-handle
**Description**

*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-asyn-co-states, but for a closed handle result is nil.

See also accepting-handle
accept-tcp-connections-creating-asyn-co-states

accepting-handle-name

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-asyn-co-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-asyn-co-states

accepting-handle-local-port

Summary Returns the local port number to which the socket in an accepting-handle was bound.
accepting-handle-local-port

**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**

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**

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:
- 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.

There is no documented return value.

See also: wait-state-collection

async-io-state Structure Type

Summary: An object that can be used to perform asynchronous I/O.

Package: comm

Slots:
  - read-status, write-status: nil for a working socket, :eof for end of file, :timeout, other values mean error.
  - user-info: Any Lisp object. LispWorks itself does not use this value for any purpose.
  - read-timeout, write-timeout: nil, or a positive real representing a time in seconds.
The COMM Package

max-read An integer specifying the maximum number of bytes to try to read between calls to the callback in `async-io-state-read-with-checking`.

old-length 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`.

name Any Lisp object. This names the state for debugging purposes.


Description The structure type `async-io-state` is an object that can be used to perform asynchronous I/O.

See also “The Async-I/O-State API” on page 363

async-io-state-abort Function

Summary Stops I/O and callbacks on an `async-io-state` and calls an abort callback.

Package `comm`
async-io-state-abort

Arguments


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 read-status.
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
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 \texttt{close-async-io-state} about accessing the state after it is closed.

See also \texttt{async-io-state-abort}
\texttt{close-async-io-state}

“The Async-I/O-State API” on page 363

\texttt{async-io-state-address} \hfill \textit{Function}

\textbf{Summary} \hspace{1cm} \textit{Returns the local address and port number for an async-io-state that has a socket.}

\textbf{Package} \hspace{1cm} \texttt{comm}

\textbf{Signature} \hspace{1cm} \texttt{async-io-state-address async-io-state => address, port}

\textbf{Arguments} \hspace{1cm} \texttt{async-io-state} \hspace{0.5cm} \textit{An async-io-state.}

\textbf{Values} \hspace{1cm} \texttt{address} \hspace{0.5cm} \textit{An integer or an ipv6-address object.}
\hspace{1cm} \texttt{port} \hspace{0.5cm} \textit{An integer.}

\textbf{Description} \hspace{1cm} The function \texttt{async-io-state-address} returns the local address and port number for \texttt{async-io-state} if it has a socket (currently all states).

\textit{address} is the local host address of the socket in the state.

\textit{port} is the local port number of the socket in the state.

See also \texttt{async-io-state-peer-address}
\texttt{get-socket-address}

“The Async-I/O-State API” on page 363
async-io-state-buffered-data-length

Summary
Returns the length of the buffered data in an async-io-state.

Package
comm

Signature
async-io-state-buffered-data-length async-io-state => length

Arguments

Values
length A non-negative integer.

Description
The function async-io-state-buffered-data-length returns the length of the buffered data in async-io-state.

See also
async-io-state-get-buffered-data
“The Async-I/O-State API” on page 363

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 363

**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 363

async-io-state-get-buffered-data

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
async-io-state
An async-io-state.

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.

Notes
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 363

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

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 363

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
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 read-timeout of `async-io-state` has passed, the `callback` is called like this:

```c
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:

```c
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 read-timeout period then state's read-status is set to `:timeout` and `callback` is called.

If `timeout` or `user-info` are supplied then they set read-timeout and `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 363
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 async-io-state)
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 reader `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.

- 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 `read-timeout` period then the callback is called with the state’s `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 the `read-timeout`, `max-read` and `user-info` slots 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 state’s `user-info`, `max-read` and `read-timeout` can be read and set 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:
(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))
      (async-io-state-finish state (+ h-end 2))
      (let ((h-object (my-parse-http-headers
          buffer 0 h-end)))
        (my-read-http-body
          state
        h-object
        (async-io-state-user-info state)))))))))

The callback is used like this:

  (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 363

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
### The COMM Package

**buffer**
A `cl:base-string` or an 8-bit `cl:_simple_array`.

**callback**
A function designator.

**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.

**needs-address**
A boolean.

**user-info**
A Lisp object.

**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.

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 363

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.
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.

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`.


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`.

See also `create-async-io-state-and-connected-udp-socket`  
`async-io-state-receive-message`  
`async-io-state-send-message-to-address`  
“The Async-I/O-State API” on page 363
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.
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.

Values None.

Description
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 348.
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`.

**Notes**

1. `async-io-state-send-message-to-address` is typically used only with an `async-io-state` containing a UDP socket, created by `create-asnc-io-state-and-udp-socket`, `create-asnc-io-state-and-connected-udp-socket` or calling `create-asnc-io-state` with `udp` non-nil.

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-asnc-io-state`, the `ipv6` argument to `create-asnc-io-state` must match the family of the socket for `async-io-state-send-message-to-address` to work.


**See also**

`create-asnc-io-state-and-connected-udp-socket`

`async-io-state-receive-message`

`async-io-state-send-message`

“The Async-I/O-State API” on page 363
async-io-state-write-buffer

Function

Summary
Asynchronously writes a buffer to an async-io-state

Package
comm

Signature
async-io-state-write-buffer 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.
- 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-write-buffer asynchronously writes the part of buffer buffer between indexes start and end to async-io-state. When this writing has succeeded or the state's write-timeout has passed, callback is called like this:

callback async-io-state buffer number-of-bytes-written

The default value of start is 0. The default value of end is the length of buffer.
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-written
```

If error-callback is nil, then callback is called, so it should check for errors using async-io-state-write-status.

If the operation does not finish within the state's write-timeout period then the state's write-status is set to :timeout and callback is called.

If timeout or user-info are supplied then they set the state's read-timeout and user-info for this and subsequent operations.

See also

- async-io-state-read-buffer
- async-io-state-read-with-checking
- “The Async-I/O-State API” on page 363

**attach-ssl**

Function

**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 => 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`.

| Values  | ssl                   | A foreign pointer of type `ssl-pointer`. |

### Description

The function `attach-ssl` attaches SSL to the socket-stream `socket-stream`.

The allowed values and meaning of the keyword arguments are as described for `socket-stream`.

Note that `attach-ssl` is used by

```lisp
(make-instance 'comm:socket-stream :ssl-ctx ...)
```

and by

```lisp
(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.

If `ssl-ctx` is a symbol, it creates the `SSL_CTX` and calls `ctx-configure-callback` if this is non-nil. If `ssl-ctx` is not a `ssl-pointer`, it creates the `SSL` object, calls `ssl-configure-callback` if this is non-nil, and sets the ACCEPT/CONNECT state if the value of `ssl-side` is not `:both`. Then it uses `SSL_set_fd` to attach the `SSL` to the socket, and records this in the socket stream. It returns the `SSL`.

The default value of `ssl-ctx` is `t` and the default value of `ssl-side` is `:server`.

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  

**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`.

See also  

`create-and-run-wait-state-collection`  
`loop-processing-wait-state-collection`  
“The Async-I/O-State API” on page 363

**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

    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 a true value 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.

See also “The Async-I/O-State API" on page 363

**close-wait-state-collection**

*Function*

**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 collection and all of its states.

See also create-and-run-wait-state-collection

“The Async-I/O-State API” on page 363

**create-and-run-wait-state-collection**

*Function*

**Summary** Creates and runs a wait-state-collection.
Package: comm

Signature: 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: wait-state-collection

A wait-state-collection.

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 (calls the function `cl:abort`).

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), 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**

1. The `wait-state-collection` does nothing by itself. You need to create and use `async-io-state` objects to actually do something.
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`

**Signature**

`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

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.

read-timeout, write-timeout and user-info set the corresponding slots in async-io-state.

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 363

**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**
```
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 ndelay name
queue-output => async-io-state
```

**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.
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 348.

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 parameters suitable for calling `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`.

`read-timeout`, `write-timeout` and `user-info` are set in `async-io-state`.

The default value of `name` is a string “Connected TCP”.

Values

- `async-io-state`  
  An `async-io-state`.

Description

- `keepalive` A generalized boolean.
- `nodelay` A generalized boolean.
- `queue-output` A boolean.
- `name` A Lisp object.
See also  
create-async-io-state  
accept-tcp-connections-creating-async-io-states  
“The Async-I/O-State API” on page 363

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, write-timeout, user-info, name  
See async-io-state for possible values.

Values

async-io-state  
An async-io-state.

Description

The function create-async-io-state-and-connected-udp-socket creates a new UDP socket, connects it to the socket address specified by hostspec and service, optionally
binds it if `local-port` or `local-address` are 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 done using `async-io-state-receive-message` and `async-io-state-send-message`.

`hostspec` and `service` are interpreted as described in “Specifying the target for connecting and binding a socket” on page 348.

`local-address` and `local-port` are also interpreted as described in “Specifying the target for connecting and binding a socket” on page 348. Both values can be `nil`.

Connecting the socket affects the destination of messages sent using the `async-io-state`, and also restricts the origin of received messages.

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 default value of `ipv6` is `:any`.

`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`.

`read-timeout`, `write-timeout`, `user-info` and `name` set the corresponding values in `async-io-state`.

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 runtime 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 363

**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`, `write-timeout`, `user-info`, `name` See `async-io-state` for possible values.
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 348. 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 runtime errors rather than signalling an error. The default value of errorp is t.

read-timeout, write-timeout, user-info and name set the corresponding values in the new async-io-state.

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` 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).

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 363

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.
- **ipv6**: `nil`, `t` or `:any`.

Values

- **handle**: A handle suitable for a `socket-stream` or a subclass, or `nil`.

Description

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 348.

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.

**See also** `ssl-pointer`

---

**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.
The function `detach-ssl` detaches the SSL from the 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.
**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**

*Function*

**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 “Loading the OpenSSL libraries” on page 360. This default may be changed by calling set-ssl-library-path.

See also openssl-version
set-ssl-library-path

get-default-local-ipv6-address

Function

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-host-entry
get-host-entry Function

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.
addrconfig A boolean.
umerichost A boolean.

Values
field-values Values, one for each field.

Description
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.

The host argument 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
The fields argument 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):

```lisp
(get-host-entry "hostname"
 :fields '(:ipv6-addresses))

(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 `numerichost` 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 `numerichost` 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  

\texttt{comm}

Signature  

\texttt{get-service-entry service protocol \&key fields => value1, value2, ...}

Arguments  

- \texttt{service}  
  An integer or a string.
- \texttt{protocol}  
  A string or \texttt{nil}.
- \texttt{fields}  
  A list of keywords specifying which information is required.

Values  

\texttt{value1, value2, ...}  
Multiple values corresponding to the keywords in \texttt{fields}, as described below.

Description  

The function \texttt{get-service-entry} looks up \texttt{service} in the system database. If \texttt{service} is an integer, it is the port number to look up. If \texttt{service} is a string, it is a name to look up (it may be one of the aliases).

If \texttt{protocol} is a string, then \texttt{get-service-entry} looks for a system database entry with protocol \texttt{protocol}, otherwise it finds the first entry with any protocol.

\texttt{fields} specifies which information is returned. When \texttt{get-service-entry} finds an entry, it returns information about it as multiple values corresponding to the keywords in \texttt{fields}. These keywords can be:

- \texttt{:name}  
  Return the name of the entry.
- \texttt{:port}  
  Return the port number of the entry.
- \texttt{:aliases}  
  Return a list of aliases of the service.
- \texttt{:protocol}  
  Return the protocol of the entry, as lower-case strings like "tcp" or "udp".

If \texttt{service} is an integer then the default value of \texttt{fields} is \texttt{(:name)}. Otherwise the default value of \texttt{fields} is \texttt{(: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 348

---

**get-socket-address**

Function

Summary
Returns the local address and port number of a given socket.

Package
```
comm
```

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.
Connected sockets have two addresses, local and remote.

See also

- `get-socket-peer-address`
- `socket-stream-address`

---

### get-socket-peer-address

**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

**Summary**

Returns the mode of the SSL.

**Package**

`comm`
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.

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**
`ipv6-address-p object => 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**
`ipv6-address-scope-id ipv6-address => scope-id`

**Arguments**

- `ipv6-address` An `ipv6-address` object.

**Values**

- `scope-id` A number or a string.

**Description**
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 an 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 `: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**

- `wait-state-collection`
  
  A *wait-state-collection*.

**Description**

The function `loop-processing-wait-state-collection` loops processing *wait-state-collection*. `loop-processing-wait-state-collection` loops waiting for any state to be ready (using `wait-for-wait-state-collection`) and processes any state that is ready (using `call-wait-state-collection`). It establishes restarts that allow aborting back into the loop, and a mechanism that allows `wait-state-collection-stop-loop` to stop the loop.
If \texttt{wait-state-collection-stop-loop} is called on the \texttt{wait-state-collection}, which can be from other threads, \texttt{wait-state-collection-stop-loop} stops looping and returns.

\textbf{Notes} \hspace{1cm} In most cases using \texttt{create-and-run-wait-state-collection} is more convenient.

\textbf{See also} \hspace{1cm} \texttt{create-and-run-wait-state-collection} \texttt{wait-state-collection-stop-loop}

\textbf{make-wait-state-collection} \hspace{1cm} \textit{Function}

\textbf{Summary} \hspace{1cm} Returns a new empty \texttt{wait-state-collection}.

\textbf{Package} \hspace{1cm} \texttt{comm}

\textbf{Signature} \hspace{1cm} \texttt{make-wait-state-collection => collection}

\textbf{Values} \hspace{1cm} \texttt{collection} \hspace{1cm} A \texttt{wait-state-collection}.

\textbf{Description} \hspace{1cm} The function \texttt{make-wait-state-collection} returns a new empty \texttt{wait-state-collection}.

\textbf{See also} \hspace{1cm} \texttt{create-and-run-wait-state-collection} \texttt{“The Async-I/O-State API” on page 363}

\textbf{open-tcp-stream} \hspace{1cm} \textit{Function}

\textbf{Summary} \hspace{1cm} Attempts to connect to a socket on a server and returns a stream object for the connection.

\textbf{Package} \hspace{1cm} \texttt{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 local-address ipv6 local-port nodelay keepalive => stream

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hostspec</td>
<td>An integer or string or an ipv6-address object.</td>
</tr>
<tr>
<td>service</td>
<td>A string or a fixnum.</td>
</tr>
<tr>
<td>direction</td>
<td>One of :input, :output or :io.</td>
</tr>
<tr>
<td>element-type</td>
<td>base-char or a subtype of integer.</td>
</tr>
<tr>
<td>errorp</td>
<td>A boolean.</td>
</tr>
<tr>
<td>read-timeout</td>
<td>A positive number, or nil.</td>
</tr>
<tr>
<td>write-timeout</td>
<td>A positive number, or nil.</td>
</tr>
<tr>
<td>timeout</td>
<td>A positive number, or nil.</td>
</tr>
<tr>
<td>ssl-ctx</td>
<td>A symbol or a foreign pointer.</td>
</tr>
<tr>
<td>ctx-configure-callback</td>
<td>A function designator or nil. The default value is nil.</td>
</tr>
<tr>
<td>ssl-configure-callback</td>
<td>A function designator or nil. The default value is nil.</td>
</tr>
<tr>
<td>local-address</td>
<td>nil, an integer, a string or a ipv6-address object.</td>
</tr>
<tr>
<td>ipv6</td>
<td>nil, t or :any.</td>
</tr>
<tr>
<td>local-port</td>
<td>nil, a string or a fixnum.</td>
</tr>
<tr>
<td>nodelay</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>keepalive</td>
<td>A generalized boolean.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>A socket-stream.</td>
</tr>
</tbody>
</table>
The function **open-tcp-stream** attempts to connect to a socket on a server and returns `stream` for the connection if successful.

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 348.

The direction of the connection is given by `direction`. Its default value is `:io`. The element type of the connection is determined from `element-type`, and is `base-char` by default.

If `errorp` is `nil`, failure to connect (possibly after `timeout` seconds) returns `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` and `ssl-configure-callback` are interpreted as described for `socket-stream`. 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 `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 369 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 ~w"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

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 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.

errorp 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.

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 and ssl-configure-callback. 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 369 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 "getDefault") of javax.net.SocketFactory (when ssl-ctx is nil) or javax.net.ssl.SSLSocket (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" verifier (Java class org.apache.http.conn.ssl.BrowserCompatHostnameVerifier). Verification with :browser-compat is a little more relaxed than with :strict.

On Android when verify is t and factory is nil, the code uses the socket factory android.net.SSLCertificateSocket-
Factory (instead of the default of javax.net.ssl.SSLSocket), which is doing the verification itself. When factory is non-nil, Android does the same as in the previous paragraph (verify using the strict verifier). The SSLCertificateSocketFactory has the advantage that it uses SNI (Server Name Indication), which makes verification work better.

When verify is a string, it has to be the hostname to use for verification, instead of the hostspec argument. The verification is done using the strict verifier.

When verify is a jobject, it must be a verifier (of class javax.net.ssl.HostnameVerifier), and it is used as-is.

The verifier classes above are part of httpclient from apache.org, and therefore to use them (which is the default when using SSL), you need to have 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 verify is nil, the hostspec is not verified, which is not recommended. However, there are valid sites which will fail verification, because they return a certificate 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 certificate 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 certificate, and in general all servers should work when using SNI. On Android the default setting uses the 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:

(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 369

`open-tcp-stream`

`openssl-version`  

Function

Summary

Returns the version of the loaded OpenSSL library.

Package  

`comm`

Signature

`openssl-version &optional what => result`

Arguments

Values

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>A string.</td>
</tr>
</tbody>
</table>

Description

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**

```r
parse-ipv6-address string &key start end trim-whitespace => result
```

**Arguments**

- **string** A string.
- **start, end** Bounding index designators of string.
- **trim-whitespace** A boolean.
Values

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>An <code>ipv6-address</code> object or <code>nil</code>.</td>
</tr>
</tbody>
</table>

Description

The function `parse-ipv6-address` parses its argument string as an IPv6 address if possible, otherwise it returns `nil`. `start` and `end` specify the subsequence of `string` to parse. The default value of `start` is 0. The default value of `end` is `nil`, meaning the length of `string`.

`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 `trim-whitespace` is `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 `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 `string` is correct, `parse-ipv6-address` constructs the `ipv6-address` object and returns it.

It does not perform any address resolution.

See also

`get-host-entry`
`string-ip-address`

### pem-read

**Function**

**Summary**

An interface to the SSL `PEM_read_bio_*` functions.

**Package**

`comm`
Signature

\texttt{pem-read thing-to-read filename &key pass-phrase callback errorp => result}

Arguments

- \texttt{thing-to-read} A string.
- \texttt{filename} A pathname designator.
- \texttt{pass-phrase} A string, or \texttt{nil}.
- \texttt{callback} A function designator, or \texttt{nil}.
- \texttt{errorp} A generalized boolean.

Values

- \texttt{result} A foreign pointer or \texttt{nil}.

Description

The function \texttt{pem-read} is an interface to the \texttt{PEM_read_bio_*} set of functions. See the manual entry for \texttt{pem} for specifications of these functions.

\texttt{thing-to-read} defines which function is required. \texttt{pem-read} concatenates \texttt{thing-to-read} with the string " \texttt{PEM_read_bio_}" to form the name of the \texttt{pem} function to call.

\texttt{filename} specifies the file to load.

If \texttt{pass-phrase} is non-nil, it must be a string, which is passed to the \texttt{pem} function. The default value of \texttt{pass-phrase} is \texttt{nil}.

If \texttt{callback} is non-nil, it must be a function with signature:

\begin{verbatim}
callback maximum-length rwflag => pass-phrase
\end{verbatim}

where \texttt{maximum-length} is an integer, \texttt{rwflag} is a boolean and \texttt{pass-phrase} is the pass-phrase to use. The default value of \texttt{callback} is \texttt{nil}, but you cannot pass non-nil values for both \texttt{pass-phrase} and \texttt{callback}.

If it succeeds, \texttt{pem-read} returns a foreign pointer to the structure that was returned by the \texttt{pem} function. If \texttt{pem-read} fails, if \texttt{errorp} is non-nil it signals an error, otherwise it returns \texttt{nil}.

The default value of \texttt{errorp} is \texttt{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`, 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 `replace-socket-stream-socket` and then using `(setf socket-stream-socket)` to set the new one is different, because the `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 \textit{socket-stream}

\textbf{server-terminate}  
\textit{Function}

\textbf{Summary} Terminates a server.

\textbf{Package} \textit{comm}

\textbf{Signature} \texttt{server-terminate \&optional process => result}

\textbf{Arguments}  
\texttt{process} A \texttt{mp:process} object or \texttt{nil}.

\textbf{Values}  
\texttt{result} A boolean.

\textbf{Description} The function \texttt{server-terminate} terminates a server process.  

If \texttt{process} is a process object it must be the result of a call to \texttt{start-up-server}. \texttt{server-terminate} terminates it, and frees all the associated resources.

If \texttt{process} is \texttt{nil} or is not supplied, the call to \texttt{server-terminate} must be inside the scope of the process that was created by \texttt{start-up-server}, which can by either \texttt{function} or \texttt{announce} that you passed to \texttt{start-up-server}. \texttt{server-terminate} returns \texttt{t} in this case, and the actual termination happens after your function (that is, \texttt{function} or \texttt{announce}) returns.

\texttt{server-terminate} returns \texttt{t} if the server was still active when it was called, otherwise it returns \texttt{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.

\textbf{Notes} In LispWorks 6.0 and earlier versions, \texttt{process-kill} is the way to terminate servers. This is deprecated, because it may leave some value in an invalid state.

See also \textit{start-up-server}
set-verification-mode

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.

:never The server will not send a client certificate request to the client, so the client will not send a certificate.

:always 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.
Once same as always except that the client certificate is checked only on the initial TLS/SSL handshake, and not again in case of renegotiation.

A list 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`.

Never 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.

Always 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 verification 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 pass-phrase 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.
- `pass-phrase` 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`, `filename`, `func` or `filename-list`.

If `dh` is non-nil, it must be a foreign pointer to a DH (corresponding to the C type `DH*`), and this DH is used as-is. The default value of `dh` is `nil`.

Otherwise, if `filename` is non-nil, it must be a pathname designator for a file containing DH parameters, which is loaded (by `read-dhparams`) and then used. In this case, `pass-phrase` and `callback` can be used, and are passed to `pem-read`.

Otherwise, if `func` is non-nil, it must be a function with signature:


func is-export keylength => dh-ptr

where is-export is a boolean, keylength is an integer, and dh-ptr is a pointer to an appropriate DH structure. `set-ssl-ctx-dh` installs `func` as the DH callback.

Otherwise (that is, if each of `dh`, `filename` and `func` are nil) then `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). `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 `read-dhparams`), and returns it. It also loads the first file of the list immediately.

`result` is `t` on success, `nil` otherwise.

See also

- pem-read
- read-dhparams
- ssl-ctx-pointer
- ssl-pointer

### set-ssl-ctx-options

**Function**

Sets the options in a `SSL_CTX`.

**Summary**

Sets the options in a `SSL_CTX`.

**Package**

comm

**Signature**

```lisp
set-ssl-ctx-options ssl-ctx &key microsoft_sess_id_bug
dscape_challenge_bug netscape_reuse_cipher_change_bug
ssref2_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:
The COMM Package

```lisp
callback maximum-length rwflag => 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**

```lisp
set-ssl-library-path library-path
```

**Arguments**

`library-path` A string or a list of strings.

**Description**
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`
socket-error

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

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.

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 \texttt{(setf \ stream\:stream-read-timeout)}. When read-timeout is \texttt{nil}, there is no timeout during reads and the call may hang.

When read-timeout is not \texttt{nil}, and there is no input from the socket for more than read-timeout seconds, any reading function returns \texttt{end-of-file}. The read-timeout does not limit the time inside \texttt{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 \texttt{(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 \texttt{(stream\:stream-read-timeout \ stream)}

The \texttt{:write-timeout} initarg specifies the write timeout in seconds, or is \texttt{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 \texttt{error} is called.

The initargs \texttt{:ssl-ctx}, \texttt{:ssl-side}, \texttt{:ctx-configure-callback} and \texttt{:ssl-configure-callback} can be be supplied to create and configure socket streams with SSL processing.

\texttt{ssl-ctx}, if non-nil, specifies that the stream uses SSL and further specifies the SSL\_CTX object to use. The value of \texttt{ssl-ctx} can be a symbol which, together with \texttt{ssl-side}, specifies which protocol to use. The value \texttt{t} or \texttt{:default} means use the default, which is currently the same as \texttt{:v23}. The values \texttt{:v2}, \texttt{:v3}, \texttt{:v23} and \texttt{:tls-v1} are mapped to the \texttt{SSL\_v2\_*}, \texttt{SSL\_v3\_*}, \texttt{SSL\_v23\_*} and \texttt{TLSv1\_*} methods respectively. With these symbol values of \texttt{ssl-ctx}, LispWorks makes a new SSL\_CTX object and uses it and frees it when the stream is closed.
The value of `ssl-ctx` can also be a foreign pointer of type `ssl-ctx-pointer` (which corresponds to the C type `SSL_CTX*`). This is used and is not freed when the stream is closed. Also an SSL object is made and used, and this object is freed when the stream is closed. The foreign pointer may be a result of a call to `make-ssl-ctx`, but it can also a result of user code, provided that it points to a valid `SSL_CTX` and has the type `ssl-ctx-pointer`.

The value of `ssl-ctx` can also be a foreign pointer of type `ssl-pointer` (which corresponds to the C type `SSL*`). This specifies the SSL to use. This maybe a result of a call to `ssl-new` but can also be the result of user code, provided that it points to a valid SSL object and has the type `ssl-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 as the `ssl-ctx` argument, it must have already been set up correctly.

`ssl-side` specifies which side the socket stream is. The value of `ssl-side` is used in two cases:

- When a new `SSL_CTX` object is created, it is used to select the method:
  
  \[
  \begin{align*}
  & :\text{client} \Rightarrow *_{\text{client\_method}} \\
  & :\text{server} \Rightarrow *_{\text{server\_method}} \\
  & :\text{both} \Rightarrow *_{\text{method}}
  \end{align*}
  \]

- 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.

If the value of `ssl-ctx` is a `ssl-pointer`, `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.

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 369 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 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 handle.

(make-instance 'comm:socket-stream
    :socket handle
    :direction :io
    :element-type 'base-char)

This example creates a socket stream with a read-timeout:
The following form illustrates character I/O in a binary socket-stream:

```
(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)))
```

The following form illustrates binary I/O in a base-char socket-stream:

```
(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`

“Socket streams with Java sockets and SSL on Android” on page 369
**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. It returns nil if SSL is not attached.

See also socket-stream
ssl-ctx-pointer

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 stream A socket stream.

Values address The remote host address of the socket stream or nil if not connected.
port The remote port number of the socket stream or nil if not connected.

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**
- `stream` A socket-stream.
- `direction` One of :input, :output or :io.
- `abort` A generalized boolean.

**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

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.
Package comm
Signature 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

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

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.
The function \texttt{start-up-server} starts a TCP server. Use \texttt{open-tcp-stream} to send messages from another client to the server.

The \texttt{function} argument provides the name of the function that processes connections. When a connection is made \texttt{function} is called with the connected socket handle, at which point you can make a stream using \texttt{make-instance} and communicate with the client. The server does not accept more connections until \texttt{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 \texttt{function} is not specified the built-in Lisp listener server is used. See the examples section below.

If \texttt{announce} is a stream or \texttt{t} (denoting \texttt{*standard-output*}), a message appears on the stream when the server is started.

If \texttt{announce} is a function it is called when the server is started. \texttt{announce} should take two arguments: \texttt{socket} and \texttt{condition}. \texttt{socket} is the socket used by the server: \texttt{announce} can therefore be used to record this socket. \texttt{condition} describes the error if there is one. \texttt{announce} can be called with \texttt{socket nil} and a condition only if \texttt{error} is \texttt{nil}. If the process is killed, \texttt{announce} is called with \texttt{socket nil} and \texttt{condition nil}.

The default for \texttt{announce} is \texttt{nil}, meaning there is no message.
service is interpreted as described in “Specifying the target for connecting and binding a socket” on page 348. 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.

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:

(comm:start-up-server :service 10243)
The COMM Package

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.

```
(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\%-" monolog)
      (format t "Self replied: \"-A\%-"
              (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**

start-up-server-and-mp is implemented only on Unix/Linux/Mac OS X platforms.

**See also**

start-up-server

**string-ip-address**

*Function*

**Summary**

Returns either an integer representing an IPv4 address or an ipv6-address object from the given IP address string.
string-ip-address  ip-address-string => ip-address

A string denoting an IP address in either dotted format for IPv4 or standard IPv6 format.

Either an integer representing an IPv4 address, or an ipv6-address object.

The function string-ip-address takes a string in the standard dotted IP address notation a.b.c.d and returns the corresponding integer IP address.

The function string-ip-address takes a string and tries to parse as an IP address. If 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 parse-ipv6-address (with trim-whitespace-p nil), which returns an ipv6-address object if it is successful or nil if it fails.

see also ip-address-string
    parse-ipv6-address

switch-open-tcp-stream-with-ssl-to-java

Make open-tcp-stream use Java sockets for SSL streams.

switch-open-tcp-stream-with-ssl-to-java &optional on

on A generalized boolean.
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 369 for details.

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`
“Socket streams with Java sockets and SSL on Android” on page 369
## wait-for-wait-state-collection

**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.

**See also**

create-and-run-wait-state-collection
loop-processing-wait-state-collection
“The Async-I/O-State API” on page 363

---

## 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 363

---

## 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 \textit{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. \texttt{with-noticed-socket-stream} is not implemented on the Windows platform.

See also \texttt{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**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Searches for interned symbols.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>common-lisp</td>
</tr>
<tr>
<td>Signature</td>
<td>apropos string &amp;optional package external-only =&gt; &lt;no values&gt;</td>
</tr>
<tr>
<td>Arguments</td>
<td>string A string designator.</td>
</tr>
<tr>
<td></td>
<td>package A package designator or nil.</td>
</tr>
<tr>
<td></td>
<td>external-only A generalized boolean.</td>
</tr>
</tbody>
</table>
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**

*Types*

**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 374

**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.

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**

**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 compila-
tion. It can have the following values. If print is nil, no infor-
mation 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 informa-
tion 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*.

eexternal-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 espe-
cially 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*. 
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 33.1 Naming conventions for FASL files

<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 33.1 Naming conventions for FASL files

<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>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>.ifasl</td>
</tr>
<tr>
<td>AIX PowerPC/64-bit LispWorks</td>
<td>.64ifasl</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>.sfasl</td>
</tr>
<tr>
<td>amd64 Solaris/64-bit LispWorks</td>
<td>.64sfasl</td>
</tr>
<tr>
<td>Intel Macintosh/32-bit LispWorks</td>
<td>.xfasl</td>
</tr>
<tr>
<td>Intel Macintosh/64-bit LispWorks</td>
<td>.64xfasl</td>
</tr>
<tr>
<td>LispWorks for iOS Runtime simulator</td>
<td>.xcfasl</td>
</tr>
<tr>
<td>LispWorks for iOS Runtime</td>
<td>.rfasl</td>
</tr>
<tr>
<td>LispWorks for Android Runtime</td>
<td>.rfasl</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
cocerce

**declaim**

*Macro*

**Summary**
Established a specified declarations.

**Package**
common-lisp
Signature: `declaim &rest declarations`

Arguments: `declarations` Declaration forms.

Description: The macro `declaim` behaves as specified in the ANSI Common Lisp Standard with one exception: for a top-level call to `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: `compile-file` `declare` `proclaim`

---

### `declare`  
**Special Form**

**Summary:** Declares a variable as special, provides advice to the Common Lisp system, or helps the programmer to optimize code.

**Package:** `common-lisp`

**Signature:** `declare declaration*`

**Arguments:** `declaration` A declaration specifier, not evaluated.

**Values:** The special form `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 takes `declare` forms and/or documentation strings.)

**Description:** There are three distinct uses of `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 runtime it also signals an error.
  
  In non-SMP LispWorks the compiler gives an error, but there is no runtime check. The runtime 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.

- **:explain** controls messages printed by the compiler while it is processing forms.

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.
- **:variables** If `optionvalue` is non-nil, list all the variables of each function, specifying whether they are floating point or not.
- **:types** If `optionvalue` is non-nil, print information about compiler transformations that depend on declared or deduced type information.
- **:floats** If `optionvalue` is non-nil, print information about calls to functions that may allocate floats.
- **:non-floats** If `optionvalue` is non-nil, print information about calls to functions that may allocate non-float numbers, for example bignums.
:all-calls If optionvalue is non-nil, print information about calls to normal functions.

:all-calls-with-arg-types
If optionvalue is non-nil, print the argument types for calls to normal functions. Must be combined with :all-calls.

:calls A synonym for :all-calls.

:boxing If optionvalue is non-nil, print information about calls to functions that may allocate numbers, for example floats or bignums.

:print-original-form
If optionvalue is non-nil, modifies the :all-calls, :floats and :non-floats explanations to include the original source code form that contains the call.

:print-expanded-form
If optionvalue is non-nil, modifies the :all-calls, :floats and :non-floats explanations to include the macroexpanded source code form that contains the call.

:print-length Use the optionvalue as the value of *print-length* for :all-calls, :floats and :non-floats explanations.

:print-level Use the optionvalue as the value of *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 86
compile
compile-file
proclaim

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 optimiza-
tion 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*
(list (find-class 'foo)
       (find-class 'bar))
       (eq (third *traced-arglist*) 'a)))
       (CLOS:SLOT-VALUE-USING-CLASS)

CL-USER 31 > (foo-a *foo*)
42

CL-USER 32 > (bar-a *bar*)
0 CLOS:SLOT-VALUE-USING-CLASS > ...
   >> CLASS           : #<STANDARD-CLASS BAR 214897F4>
   >> CLOS::OBJECT    : #<BAR 2148820C>
   >> CLOS::SLOT-NAME : A
0 CLOS:SLOT-VALUE-USING-CLASS < ...
   << VALUE-0 : 99
99

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

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

defined-package-name
A string designator.

option
Keyword options.

add-use-defaults
A keyword

Values

package
A package.

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

(defpackage "MY-PACKAGE" (:use "CAPI")
 (:add-use-defaults t))

(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.
### Values

<table>
<thead>
<tr>
<th>test</th>
<th>Filtering test (only pathnames matching the test are collected).</th>
</tr>
</thead>
<tbody>
<tr>
<td>directories</td>
<td>A boolean controlling whether non-matching directories are included in the result.</td>
</tr>
<tr>
<td>flat-file-namestring</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>link-transparency</td>
<td>If <code>nil</code>, then symbolic links are not followed. This means that returned names are not necessarily truenames, but has the useful feature that the <code>pathname-directory</code> of each pathname returned is the directory supplied as argument. The default value of <code>link-transparency</code> is given by the special variable, <code>*directory-link-transparency*</code>, which has initial value <code>t</code> on UNIX/Linux/Mac OS X. By setting this variable to <code>nil</code>, you can get the old behavior of <code>directory</code>. On Windows, where the file system does not normally support symbolic links, this variable is initially <code>nil</code>.</td>
</tr>
<tr>
<td>non-existent-link-destinations</td>
<td>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 <code>:link-transparency</code> is non-nil and <code>:non-existent-link-destinations</code> is <code>nil</code> (this is the default on UNIX/Linux/Mac OS X), then symbolic links to nonexistent files do not appear. The default value is <code>nil</code>.</td>
</tr>
<tr>
<td>pathnames</td>
<td>A list of physical pathnames.</td>
</tr>
</tbody>
</table>
**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
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

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 :unspecifitc
      :directory (list :absolute "tmp")
      :name :unspecifitc
      :type :unspecifitc
      :version :unspecifitc))
(#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  (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:

```
documentation (dspec t) (doc-type (eql 'dspec:dspec))

(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:

```
(documentation (dspec t) (doc-type (eql 'def-saved-value))
```

This allows LispWorks IDE commands such as Expression > Documentation to display the documentation.

**double-float**

*Type*

A subtype of float.
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* Variable

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.

:solaris2 Solaris2
:aix AIX PowerPC
:svr4 System 5 Release 4 machine (for example Solaris2)
: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 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:
Conditionalization for LispWorks versions

The following features can be used to distinguish between versions of LispWorks:

:lisplworks4 All major version 4 releases.
:lisplworks4.4 Release 4.4.x
:lispworks5  All major version 5 releases.
:lispworks5.0  Release 5.0.x
:lispworks5.1  Release 5.1.x
:lispworks6  All major version 6 releases.
:lispworks6.0  Release 6.0.x
:lispworks6.1  Release 6.1.x
:lispworks7  All major version 7 releases.
:lispworks7.0  Release 7.0.x

Code using new LispWorks functionality should be conditionalized only using features representing earlier versions, so as to future-proof your code:

(defun *feature-added-in-LispWorks-7.0*
  #+(or lispworks4 lispworks5 lispworks6) nil
  #-(or lispworks4 lispworks5 lispworks6) t)

This is because a feature added in LispWorks 7.0 will generally also be in LispWorks 7.1, LispWorks 8.0 and all later versions.

Similarly:

(defun *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:

(defun *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 conditional-
ization 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.
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`.

There is an example in “Stream directionality” on page 340.

The predicate `interactive-stream-p` is implemented as a generic function. The `fundamental-stream` class provides a default method that returns `nil`.

### interactive-stream-p

**Summary**

A generic function that determines if an object is an interactive stream.

**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**
c1

**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

**Type**

**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**

**Function**

**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 Chap-
ter 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 "~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 "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.

**Description**
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.

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.
The COMMON-LISP Package

: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 135

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

\texttt{make-hash-table \&key test size rehash-size rehash-threshold hash-function weak-kind single-thread free-function => hash-table}

Arguments

- \textit{test} A designator for a function of two arguments, which returns \texttt{t} if they should be regarded as the same and \texttt{nil} otherwise.

- \textit{hash-function} A designator for a function of one argument, which returns a hash value.

- \textit{weak-kind} \texttt{t}, \texttt{nil}, or one of the keywords \texttt{:value}, \texttt{:key}, \texttt{:both}, \texttt{:one} and \texttt{:either}.

- \textit{single-thread} A generalized boolean.

- \textit{free-function} A designator for a function of two arguments.

Description

The standard definition of \texttt{make-hash-table} is extended such that \textit{test} can be any suitable user-defined function, except that it must not call \texttt{process-wait} or similar \texttt{mp} package functions which suspend the current process. If \textit{test} is not one of the standard test functions (\texttt{eq, eql, equal and equalp}), and if \textit{hash-function} is not supplied, then the hash value is the same as would be used if \textit{test} were \texttt{equalp}.

\textit{hash-function} may be supplied only if \textit{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 \textit{weak-kind} is non-nil, it makes \textit{hash-table} weak. Its semantics are the same as the second argument of \texttt{set-hash-table-weak}, that is:

\begin{verbatim}
(make-hash-table :weak-kind weak-kind <other-args>)
\end{verbatim}

is equivalent to

\begin{verbatim}
(let ((ht (make-hash-table <other-args>)))
  (set-hash-table-weak ht weak-kind)
  ht)
\end{verbatim}

The default value of \textit{weak-kind} is \texttt{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 automatically 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 135

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  make-sequence result-type size &key initial-element => sequence
Arguments  result-type  A type specifier.
size  A non-negative integer.
initial-element  An object.
Values  sequence  A sequence.
Description  The function make-sequence 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
map
merge

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**

Function

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.
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`

**Function**

**Summary**
Redefines the standard `merge` function allowing it to take any type specifier.

**Package**
`common-lisp`

**Signature**
```lisp
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 :s01-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 analyses 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 :output 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

Generic Function

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.

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 340.

See also

fundamental-output-stream
input-stream-p

proclaim

Function

Summary

Established a specified declaration in the global environment.

Package

common-lisp

Signature

proclaim declaration-list => nil

Arguments

declaration-list A list of declaration forms to be put into immediate and pervasive effect.

Values

Returns nil.

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 86 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.

`(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`
- `declaim`
- `declare`
### restart-case

**Macro**

**Summary**
Evaluates a restartable form in a special dynamic environment.

**Package**
common-lisp

**Signature**
```
restart-case restartable-form {clause} => result*
```

**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 storage 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 storage, 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.

**Notes**

1. The segments information is useful for debugging problems with memory management, but for analysis of application allocation `(room :full)` gives enough infor-
mation. 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).

2. See “Memory Management in 64-bit LispWorks” on page 128 for a description of allocation types and segments.
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 112K

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,
    Awaiting promotion = 23K, sweeps = 10
  Segment 222B4498: Total Size 3886K, Allocated 650K, Free 3232K
    minimum free space 0K,
    Awaiting promotion = 51K, sweeps = 2
  Segment 2070DC18: Total Size 68K, Allocated 64K, Free 0K
    minimum free space 3K,
    Awaiting promotion = 0K, sweeps = 4
  Segment 21D84498: Total Size 1088K, Allocated 613K, Free 470K
    minimum free space 0K,
    Awaiting promotion = 0K, sweeps = 4
  Segment 200528D8: Total Size 240K, Allocated 118K, Free 118K
    minimum free space 0K, static
  Segment 21E94498: Total Size 4224K, Allocated 32591K, Free 6461K
```
2107K, Free 2111K
    minimum free space 0K,
    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 0K,
    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 storage management system” on page 112

---

**short-float**

**Type**

**Summary**

A subtype of float.

**Package**

common-lisp

**Signature**

short-float
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.

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

Identifies the physical location of the computer.

package common-lisp

short-site-name => description
(setf short-site-name) description => description

description A string or nil.

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
single-float

Summary A subtype of float.

Package common-lisp

Signature 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 Lisp-Works 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

Summary Identifies the Operating System.

Package common-lisp

Signature software-type => description
The function `software-type` returns a string representing a generic name of the Operating System, or `nil` if this cannot be determined.

On Microsoft Windows 98 and Millennium, `software-type` returns "Windows". On Windows 8, Windows 7 and Windows Vista, `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 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".

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 *background-* 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.
Step up out of this form without stepping its subforms.

Return a value to use for this form.

Quit from the current stepper level.

Redo one of the previous commands.

Get an item from the history list and put it in a variable.

List available commands.

Replace one form with another form in previous command and redo it.

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.

The following examples illustrate some of these commands.

```lisp
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
 1 -> :s
 1
 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
 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:

```
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.
Values

<table>
<thead>
<tr>
<th>Values</th>
<th>type</th>
<th>A type specifier.</th>
</tr>
</thead>
</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 340.

See also

- `buffered-stream`
- `fundamental-binary-stream`
- `fundamental-character-stream`

---

**string**

Type

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 *.

---

628
Description

The union of all string types as specified in the standard, but extended with an extra parameter: \textit{element-type}.

\texttt{(string length element-type)} means all string types whose element type is a subtype of \textit{element-type}. That is:

\begin{align*}
\text{(string * base-char)} & = \text{(vector base-char *)} \\
\text{(string * lw:bmp-char)} & = \text{(or (vector base-char *) (vector lw:bmp-char *))} \\
\text{(string * character)} & = \text{(or (vector base-char *) (vector lw:bmp-char *) (vector character *))}
\end{align*}

Example

\begin{verbatim}
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
\end{verbatim}

See also

*\texttt{*default-character-element-type*}

set-default-character-element-type

\textbf{time} \hspace{1cm} \textit{Macro}

Summary

Determines the execution time of a form in the current environment.

Package

common-lisp

Signature

time form => result

Arguments

\textit{form} \hspace{1cm} A form to be evaluated.

Values

\textit{result} \hspace{1cm} The result of the evaluation of the form.

Description

time can be used to determine execution times. The macro evaluates the form \textit{form} and returns its value \textit{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:
   
   `(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

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   = 49320222956 bytes
0 Page faults
Calls to %EVAL 72000048
3.460518E9

See also
extended-time
with-other-threads-disabled
with-unique-names
“Guidance for control of the storage management system” on page 112

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 ::= (dspec {keyword form}* )

dspec ::= function-name | (method generic-function-name [qualifier] (class* ) )


qualifier ::= :after | :before | :around

function-name ::= symbol | (setf symbol)

631
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.

- **dspec**: Specifies the functional definition which is to be traced. This either has the same form as above, or specifies a method by the name of its generic function and by a list of classes to specialize the arguments to the method. In this latter case the list of classes must correspond exactly to the classes of the specialized parameters of an existing method, and then only this method is traced (as opposed to the corresponding generic function).

- **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,

```lisp
(trace (print :entrycond nil
             :exitcond nil
             :allocation $$\text{print-allocation}$$))
```
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 allocation. 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 message 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 format 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 followed 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 output 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 different 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  

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 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 47.
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.
CL-USER 2 > (defun outer ()
   (inner))
OUTER

CL-USER 3 > (defun inner ()
   10)
INNER

CL-USER 4 > (trace (inner :inside outer))
   ;; only trace when inside OUTER
(INNER)

CL-USER 5 > (inner)
   ;; no tracing occurs since we are not inside OUTER
   10

CL-USER 6 > (outer) ;; INNER is traced inside OUTER
   0 INNER > NIL
   0 INNER < (10)
   10

CL-USER 7 >

Example 3
To trace a method:

(defmethod foo (x) x)
(trace ((method foo (t))))

Example 4
To trace a setf function:

CL-USER 56 > (defvar *a* 0)
   *A*

CL-USER 57 > (defun (setf foo) (x y) (set y x))
   (SETF FOO)

CL-USER 58 > (trace (setf foo))
   ((SETF FOO))

CL-USER 59 > (setf (foo 'a*) 42)
   0 (SETF FOO) > (42 *A*)
     >> X : 42
     >> Y : *A*
   0 (SETF FOO) < (42)
   42
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*

**Summary**  
Returns the truename of a pathname.

**Package**  
common-lisp

**Signature**  
truename filespec => truename

**Arguments**  
filespec  
A pathname designator.

**Values**  
truename  
A fully-specified physical pathname.

**Description**  
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
*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 + - / *))
*


USER 13 > (+ 2 3)
0 + > (2 3)
0 + < (5)
5

USER 14 > (untrace +)
(* |/|)

USER 15 > (+ 2 3)
5

To untrace a method:

(untrace (clos:method foo (t)))

See also
trace
untrace-new-instances-on-access
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`
The COMPILER Package

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**
```
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 \texttt{compiler::%pass%}, in which case no transformation is applied).

Values  \textit{list-of-transforms}  A list of the names of transforms defined for the function, including the one just added.

Description  \texttt{deftransform}, like \texttt{defmacro}, defines a function that computes the expansion of a form. Transforms are only used by the compiler and not by the interpreter. \texttt{deftransform} allows you to add to the optimizations performed by the compiler.

Examples  

\begin{verbatim}
(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 d 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
\end{verbatim}
(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 arguments 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 preserves 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
The COMPILER Package
The DBG Package

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”.

*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)
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 objects 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)
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.

**executable-log-file**

*Function*

**Summary**
Returns the default bug form log file.

**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

<table>
<thead>
<tr>
<th>Summary</th>
<th>A list of packages whose symbols should not be displayed in debugger output.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>dbg</td>
</tr>
<tr>
<td>Initial value</td>
<td>A list containing the dbg and conditions packages.</td>
</tr>
<tr>
<td>Description</td>
<td>dbg:<em>hidden-packages</em> 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.</td>
</tr>
</tbody>
</table>
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")

CL-USER 4 : 1 > :b 3
Call to CAPI::CAPI-TOP-LEVEL-FUNCTION
Call to CAPI::INTERACTIVE-PANE-TOP-LOOP
Call to MP::PROCESS-SG-FUNCTION

CL-USER 5 : 1 >

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
### log-bug-form

**Function**

**Summary**

Writes a log of an error. This is useful in an application's error handlers.

**Package**

`dbg`

**Signature**

```lisp
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.

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.

**See also**

executable-log-file
logs-directory

---

**logs-directory**  
*Function*

**Summary** Returns the directory in which LispWorks puts log files.

**Package** dbg

**Signature** logs-directory => dir

**Values** dir A directory pathname.

**Description** 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

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*

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)
Variable

See also

set-debugger-options

*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.
The DBG Package

*print-restart-frames*

Variable

<table>
<thead>
<tr>
<th>Summary</th>
<th>Controls whether restart frames are printed in debugger output.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>dbg</td>
</tr>
<tr>
<td>Initial value</td>
<td>nil</td>
</tr>
<tr>
<td>Description</td>
<td>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 <code>*print-restart-frames*</code> is set to <code>t</code> then the restart frames are shown.</td>
</tr>
</tbody>
</table>

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.
See also set-debugger-options

set-debugger-options Function

Summary Sets debugger printing control variables.

Signature set-debugger-options &key all bindings catchers hidden handler restarts invisible

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>bindings</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>catchers</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>hidden</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>handler</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>restarts</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>invisible</td>
<td>A generalized boolean.</td>
</tr>
</tbody>
</table>

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*
The call frames are always displayed, so there is no option to control that.

See also  set-debugger-options

**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 multiprocess 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
(setq 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
Signature  with-debugger-wrapper  wrapper &body body => results

Arguments  
wrapper  A function designator.
body  Forms.

Values  results  Results of body.

Description  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.
;;; application code
(in-package "CL-USER")

(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
(defglobal-parameter *my-debugger-lock*
  (mp:make-lock :name "Debugger Lock"))

(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
The DBG Package
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:
The internal database of definitions performed in this image.

Prompt for a tags file, when first used.

Either a tags file or a tags database.

A tags database is a fasl 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

**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`.

---

Macro
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 define-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

Informs 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`

define-dspec-class

Macro

Defines a dspec class.

Package `dspec`

Signature

```
define-dspec-class name superspace documentation &key pretty-name undefiner canonicalize prettify definedp object-dspec defined-parts aggregate-class
```

Arguments

- `name`: A symbol naming the dspec class.
- `superspace`: A symbol naming the superspace.
- `documentation`: A string describing the dspec class.
- `undefiner`: A function that generates the undefining form for the class.
- `canonicalize`: A function to canonicalize a dspec if it belongs to the class.
**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, providing handlers for definitions in that dspec class.

`define-dspec-class` defines `name` as a dspec class, inheriting 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 omitted 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

(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 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-
wise. For example, the object-dspec function for package
dspecs might be:

#'(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 61.

**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`
Signature  
\texttt{define-form-parser definer-and-options \&optional parameters \&body body \=} \rightarrow \texttt{parser}

Arguments  
definer-and-options  
A symbol \textit{definer} naming a definer of functions, macros, variables and so on, or a list (\textit{definer options}) where \textit{options} is a plist of keys and values.

\textit{parameters}  
nil, or list of parameters \textit{params} in the top level form, optionally ending with \texttt{\&rest param-getter}.

\textit{body}  
The body of a parser function.

Values  
\texttt{parser}  
A form parser function.

Description  
The macro \texttt{define-form-parser} defines a form parser for forms beginning with \textit{definer}.

\textit{options} is a property list with the following keys allowed:

\texttt{\:parser}  
A parser function \texttt{parser-function}.

\texttt{\:alias}  
A dspec class or alias \texttt{alias}.

\texttt{\:anonymous}  
A boolean.

The parser function defined is named by \texttt{parser-function}. If the \texttt{\:parser} option is omitted then the name defaults to a symbol in the current package whose symbol name is the symbol name of \textit{definer} with "\texttt{-FORM-PARSER}" appended.

If \texttt{parameters} and \texttt{body} are given, then \texttt{parser-function} is defined as a global function that is expected to return a dspec for the defining form or \texttt{nil} if this is not possible. Within \texttt{body}, \textit{definer} is bound to the \texttt{car} of the actual form being parsed. In simple cases, this is just \textit{definer}, but if the form parser is used as in the \texttt{\:alias} option of another form parser then the symbol will be bound to the \texttt{car} of that form instead.
The \textit{params} are bound to subsequent subforms of the defining form. If \texttt{&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 \textit{parameters} and \textit{body} are omitted, then \textit{parser-function} is expected to be a form parser defined by a \texttt{define-form-parser} form, or you can specify as an alias a definer with an existing form parser via the value \textit{alias} of the \texttt{:alias} key in \textit{options}.

If the \texttt{:anonymous} option is non-nil then \textit{definer} is not associated with the form parser. This is useful in conjunction with \textit{parameters} and \textit{body} for defining generic form parsers that can be used in other \texttt{define-form-parser} forms.

LispWorks contains pre-defined form parser functions for the Common Lisp definers \texttt{defun, defmethod, defgeneric, defvar, defparameter, defconstant, defstruct, defclass, defmacro} and \texttt{deftype} and for LispWorks definers such as \texttt{fli:define-foreign-type} and \texttt{dspec:define-form-parser} itself.

When a defining symbol \textit{definer} has an associated form parser, this parser function is used by the source location commands such as \texttt{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.

\textbf{Example}

Define a parser for \texttt{def-foo} forms which have a single name as the second element in the form:

\begin{verbatim}
(dspec:define-form-parser def-foo (name)
  `(def-foo ,name))
\end{verbatim}

Define a parser for \texttt{def-other-foo} forms which are like \texttt{def-foo} forms:
(defother-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:

(dspect: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:

(dspect: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))

(dspect:define-dspec-alias my-defmethod
   (name &rest args)
 `,(defmethod ,name ,@args))

(my-defmethod foo ((x number))
  42)

(dspect:define-form-parser
   (my-defmethod
    (:parser
     #.(dspect:get-form-parser 'defmethod))))

A simpler way to write the last form is:

(dspect:define-form-parser
   (my-defmethod
    (:alias defmethod)))
**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:

```lisp
(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 function `dspec-class` returns the dspec class name for `dspec`.

**Example**

```lisp
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
```

See also `dspec-name`
**dspec-defined-p**  

**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 \textit{dspec} is a local dspec, the parent function is located.

\textbf{Example}

\begin{verbatim}
CL-USER 6 > (dspec:dspec-definition-locations 
  '(defun foo-bar)) 
  (((DEFSTRUCT FOO) #P"C:/temp/hack.lisp"))
\end{verbatim}

\textbf{See also}

name-definition-locations

\section*{dspec-equal \textit{Function}}

\textbf{Summary}

Tests two dspecs for equality as dspecs.

\textbf{Package}

dspec

\textbf{Signature}

dspec-equal \textit{dspec1} \textit{dspec2} \Rightarrow \textit{result}

\textbf{Arguments}

dspec1, dspec2 Dspecs.

\textbf{Values}

\textit{result} A boolean.

\textbf{Description}

The function \textit{dspec-equal} compares \textit{dspec1} and \textit{dspec2} for equality as dspecs.

Both arguments are canonicalized before the comparison.

Dspects in different subclasses of the same namespace are \textit{dspec-equal} if their names match.

Unknown dspecs are compared simply by \textit{equal}.

\textbf{Example}

\begin{verbatim}
CL-USER 44 > (dspec:dspec-equal '(deftype foo) 
  '(defclass foo)) 
  T
\end{verbatim}

\section*{dspec-name \textit{Function}}

\textbf{Summary}

Extracts the name from a 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   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    (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   dspec-subclass-p class1 class2 => result

Arguments   class1, class2    Symbols naming dspec classes.
Values

result  A boolean.

Description

The function \texttt{dspec-subclass-p} determines whether the dspec class denoted by \textit{class1} is a subclass of that denoted by \textit{class2}.

Example

\begin{verbatim}
CL-USER 55 > (dspec:dspec-subclass-p 'defmacro 'type)
NIL

CL-USER 56 > (dspec:dspec-subclass-p 'defmacro 'function)
T
\end{verbatim}

\textbf{dspec-undefiner} \quad \textit{Function}

\textbf{Summary}

Returns an undefining expression for a dspec.

\textbf{Package}

dspec

\textbf{Signature}

dspec-undefiner dspec => form

\textbf{Arguments}

dspec  A dspec.

\textbf{Values}

form  A Lisp form.

\textbf{Description}

The function \texttt{dspec-undefiner} returns a form which would undefine dspec, whether or not \texttt{dspec} is currently defined. If no such form can be constructed, \texttt{nil} is returned.

Example

\begin{verbatim}
CL-USER 66 > (dspec:dspec-undefiner '(defun foo))
(PROGN (FMAKUNBOUND (QUOTE FOO)) (SETF (DOCUMENTATION (QUOTE FOO) (QUOTE FUNCTION)) NIL))
\end{verbatim}

\textbf{find-dspec-locations} \quad \textit{Function}

\textbf{Summary}

Returns the locations of the definitions of a dspec.
The function \texttt{find-dspec-locations} returns the locations of the relevant definitions for the dspec \texttt{dspec}.

For each known definition \texttt{recorded-dspec} names the definition that defined \texttt{dspec} in \texttt{location}, and \texttt{location} is a pathname or keyword as described in \texttt{at-location}.

If \texttt{dspec} is a local dspec, the parent function is located.

The location information is collected from all finders on \texttt{*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 (\texttt{recorded-dspec location}), as compared by \texttt{dspec-equal} and location equality, then only the first occurrence of the pair (in the order of \texttt{*active-finders*}) appears in \texttt{locations}.

\textbf{See also} \texttt{*active-finders*}  
\texttt{dspec-definition-locations}  
\texttt{dspec-equal}

\textbf{Function} \texttt{find-name-locations}  
\textbf{Summary} Returns the locations of the definitions of a name.
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** define => parser

**Arguments**

- **define**: 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.

```lisp
CL-USER 1 > dspec:get-form-parser 'defun
DSPEC:NAME-ONLY-FORM-PARSER
```

See also `define-form-parser` `parse-form-dspec`

---

**local-dspec-p**

**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))))

(name-defined-dspecs classes name => dspecs)

See also at-location
defrecord-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

describe

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
contributes one subform and returns it.

name-only-form-parser can be used for function defini-
tions 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))

(dssec:define-form-parser
  (my-definer (:parser
    dspec:name-only-form-parser)))
See also  \texttt{define-form-parser}

\textbf{parse-form-dspec}  \textit{Function}

\textbf{Summary} PARSES THE DSPEC FROM A DEFINING FORM.

\textbf{Package} dspec

\textbf{Signature} \texttt{parse-form-dspec form \rightarrow result}

\textbf{Arguments} \textit{form} A form.

\textbf{Values} \textit{result} A dspec or \texttt{nil}.

\textbf{Description} The function \texttt{parse-form-dspec} invokes the defined form parser for \textit{form} and returns the resulting dspec.

\textbf{Example} \begin{verbatim}
(parse-form-dspec '(def-foo my-foo (arg) (foo-it arg))) =>
(def-foo my-foo)
\end{verbatim}

See also \texttt{define-form-parser}
\texttt{get-form-parser}

\textbf{record-definition}  \textit{Function}

\textbf{Summary} CHECKS FOR EXISTING DEFINITIONS AND RECORDS A NEW DEFINITION.

\textbf{Package} dspec

\textbf{Signature} \texttt{record-definition dspec location &key check-redefinition-p \rightarrow result}

\textbf{Arguments} \textit{dspec} A dspec.

\textit{location} A pathname or keyword.
check-redefinition-p

A boolean.

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 record-definition tells the system that dspec is defined at location.

The system-provided definer macros call the function record-definition with the current location.

location should be a pathname or keyword as returned by location.

When check-redefinition-p is true, it checks for existing definitions and reports these according to the value of *redefinition-action*. The default value of check-redefinition-p is t.

If the definition is made, then result is true. If the definition is not made then result is nil. This can happen if you choose the "Don’t redefine ..." restart at a redefinition error.

Notes

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.

Compatibility

record-definition was documented in the 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 dspec package.

See also

define-dspec-class
*redefinition-action*

location

“Recording definitions and redefinition checking” on page 69
The DSPEC Package

*record-source-files*  Variable

Summary  Controls whether the locations of definitions are recorded.

Package  dspec

Initial value  t

Description  The variable *record-source-files* controls whether locations of definitions are recorded in the internal tags database.

Compatibility notes  *record-source-files* 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.

See also  *active-finders*

*redefinition-action*  Variable

Summary  Specifies the action on some redefinitions.

Package  dspec

Initial value  :warn

Description  *redefinition-action* controls messages about redefinitions seen by the source location system.

If *redefinition-action* 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`.

**Notes**
*redefinition-action* does not affect the behavior of `cl:defstruct`.

**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 equiv-
alented 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:

(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`

**single-form-form-parser**

*Function*

**Summary**
A pre-defined form parser.

**Package**
dspec

**Signature**
`single-form-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 `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 `defsystem`. 
You can define it to be the parser for your defining forms.
using `define-form-parser`.

See also `define-form-parser`

**single-form-with-options-form-parser**

*Function*

**Summary**
A pre-defined form parser.

**Package**
dsputc

**Signature**
single-form-with-options-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 `single-form-with-options-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 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 `defstruct`.


You can define it to be the parser for your defining forms.
using `define-form-parser`.
traceable-dspec-p

Summary Tests whether definition can be traced.

Package dspec

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

Summary Gets and sets the global tracing state.

Package dspec

Signature tracing-enabled-p => 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`

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 are 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 383
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.

: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.

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 383
The EXTERNAL-FORMAT Package

:utf-16
:utf-16be
:utf-16le
:utf-16-native
:utf-16-reversed

External Formats

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 are 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 383

:utf-32
:utf-32le
:utf-32be
:utf-32-native
:utf-32-reversed

External Formats

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
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 are 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.
See also “16-bit External formats guide” on page 383

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.

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)

:jis 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

de 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`
find-external-char

Function

Summary
Returns the character of a given code in a specified character set.

Package
external-format

Signature
find-external-char code set => char

Arguments
- code: A character code. This is an integer.

Values
- char: The character represented by code. If code is not a legal code in the specified set, the return value is undefined.

Description
Returns the character that has the code code (an integer) in the coded character set specified by set, or 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 ef-spec An external format spec.

env An environment across which the spec should apply.

Values result A boolean.

Description This predicate tests whether the external format spec given in ef-spec is valid (in the environment env).

result is t if ef-spec is a valid spec, and nil otherwise.

Example (valid-external-format-p '(:Unicode :eol-style :lf))
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.

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

Functions

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**

*Function*

**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**

Adds a symbol to the list of profiled symbols.

**Package**

hcl

**Signature**

`add-symbol-profiler symbol => nil`
Arguments  

symbol  
A symbol to be added to the *profile-symbol-list*.

Values  

Returns nil.

Description  

add-symbol-profiler adds a symbol to *profile-symbol-list*, the list of profiled symbols.

See also  

*profile-symbol-list*  
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 storage 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 117

---

**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.
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.

Values        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 95.

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 cl:format. It does not affect any of the I/O in body. The default value of stream is t, meaning standard output.

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 hcl:special-dynamic to cl:special)

The inconsistent messages are the most useful. A well written program should not produce any such message.

unused controls whether to report symbols that are proclaimed special but are otherwise not used. For this option to be really useful, body needs to force compile many source files.

Since such unused variables do not affect the code, 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)

only-bound controls whether to report symbols that have been seen bound, but whose value has not been read. The comments about unused also apply to only-bound.

wrong-global controls whether to print symbols that are bound but are also proclaimed hcl:special-global. If the proclamation preceded the binding, the compiler will signal a compiler-error.

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 defglobal-parameter and defglobal-variable), or proclaimed 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`
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`.

See also

`*android-main-process-for-testing*`
`android-main-thread-p`
Chapter 16, “Android interface”

### 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-`
The HCL Package

`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-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.

**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 array-weak-p returns t if its argument object is a weak array, and otherwise returns nil.

See also  make-array
set-array-weak

augment-environment

Function

Summary  Returns a new environment based on an existing one with different bindings.

Package  hcl

Signature  augment-environment env &key variable symbol-macro function macro declare reset => newenv

Arguments  env  An environment or nil
The function augment-environment returns a new environment newenv, based on env but modified according to the keyword arguments variable, symbol-macro, function, macro, declare and reset.

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 \((\text{symbol macrofunction})\) and \(\text{newenv}\) 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 \(\text{newenv}\) 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.

\(\text{newenv}\) has the same extent as \(\text{env}\), that is it might have dynamic extent within the function that created \(\text{env}\).

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
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 implementations. 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 117

*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*.
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.

Compatibility note

These variables are 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 `progn`.

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, \((\text{gc-generation t})\)
will free a large amount of space in generation 2.

\textbf{block-promotion} can be thought of as doing \texttt{set-promotion-count}
on generation 1 with an infinite \texttt{count}, for the duration of body.

\textbf{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 LispWorks \texttt{compile-file}, to ensure the transient compile-time data gets collected.

\textbf{block-promotion} has global scope and hence may not be useful in an application such as a multithreaded server. During the execution of \texttt{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.

\textbf{Notes}

1. Symbols and process stacks are allocated in generation 2 or 3 (see \texttt{*symbol-alloc-gen-num*}) hence \textbf{block-promotion} cannot prevent these getting into that generation. \texttt{allocation-in-gen-num} can also cause allocation in higher generations.

2. In 64-bit LispWorks, \textbf{block-promotion} is implemented using \texttt{set-blocking-gen-num}.

\textbf{See also}

\texttt{allocation-in-gen-num}
\texttt{mark-and-sweep}
\texttt{set-promotion-count}

\textbf{building-universal-intermediate-p} \hspace{1em} \textit{Function}

\textbf{Summary} Deprecated. Simply returns \texttt{nil}.

\textbf{Package} \texttt{hcl}
**building-universal-intermediate-p**

**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.

**Values**

`callees`  A list.

**Description**

The function `calls-who` returns a list of the dspects 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**
`change-directory` changes the current directory to that specified by `directory`. `directory` may be an absolute or relative pathname.

Use `get-working-directory` to find the current directory.

**Notes**
`change-directory` 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 storage management system” on page 112

---

**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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>full</td>
<td>A generalized boolean.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-size</td>
<td>A positive integer.</td>
</tr>
</tbody>
</table>

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. **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 storage management system” on
page 112

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 gen-
eration 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`
`set-promotion-count`
“Memory Management in 32-bit LispWorks” on page 117

### clear-code-coverage

### reset-code-coverage

### restore-code-coverage-data

#### Functions

| 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. |
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.

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**

A structure containing information about code coverage.
<table>
<thead>
<tr>
<th>Package</th>
<th>hcl</th>
</tr>
</thead>
</table>
| 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” |
code-coverage-data-generate-coloring-html  Function

Summary  Generates HTML showing the code coverage.

Package  hcl

Signature  code-coverage-data-generate-coloring-html target &key
           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  target  A pathname designator.
            code-coverage-data  A code-coverage-data object.
            shared-source-directory  A pathname designator.
            filter  A string, a function or a symbol naming a
                   function.
            target-type  A string or nil.
            color-uncovered  A boolean.
            color-covered  A boolean.
            show-counters  A boolean.
            counter-space  nil, :before, :after, :both or t.
            index-filename  A pathname designator or nil.
            index-name  A string.
            index-sort  One of the keywords :relative-name,
                         :name and :uncovered.
            index-mark-not-entered  A boolean.
            index-mark-partial
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 runtime 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 runtime part for uncovered lambdas. The default value of index-mark-not-entered is t.

index-mark-partial controls whether to mark cells in the runtime 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-runtime 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-background-colors. 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: runtime lambdas and optional non-runtime lambdas. "Lambda" here means a separate piece of code (for example code that is called inline does not count as a separate lambda). Runtime lambdas refer to code that is expected to run at runtime, which includes things like functions and methods. Non-runtimes are other lambdas, like macros and top-level forms (as known as one-shot forms). More accurately, runtime and non-runtime 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 runtime and the non-runtime 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 runtime columns, Partial and None cells which are non-zero are optionally marked with a colored background. This helps you to see which files contain runtime 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 106.

#### 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`
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**

<table>
<thead>
<tr>
<th>Summary</th>
<th>A structure containing code coverage statistics.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>hcl</td>
</tr>
<tr>
<td>Readers</td>
<td>code-coverage-file-stats-source-file</td>
</tr>
</tbody>
</table>
| Pseudo-readers | 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

```lisp
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** Macros and macro-like (for example `cl:defsetf`).
- **:one-shot** Load time lambdas that the compiler generates.
- **:lambdas** Other lambdas (including `cl:defmethod`).

In addition, the following three values of `keyword` can be
used:

- **:all** 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 runtime/non-runtime distinction is intended to correspond to code that would run in the actual application (runtime) 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 "runtime" and have been called.

See also Chapter 10, “Code Coverage”

code-coverage-data-generate-statistics

code-coverage-set-editor-colors

Function

Summary
Specifies the colors that the editor uses to color code coverage.

Package
hcl

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``

### code-coverage-set-editor-default-data

**Function**

**Summary**
Sets the default code coverage data that the editor uses when coloring.

**Package**
hcl

**Signature**
`code-coverage-set-editor-default-data object`

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>A <code>code-coverage-data</code> object, a string, a pathname or nil.</td>
</tr>
</tbody>
</table>

**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` `error` `warn` `eliminated` `marked-cell`

**Arguments**

- `counters` A string.
- `counters-negative` A string.
- `uncovered` A string.
- `partially-covered` A string.
- `fully-covered` A string.
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 106 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 106

---

**collect-generation-2**

*Function*

**Summary** Controls whether generation 2 is garbage collected in 32-bit LispWorks.

**Package** `hcl`
The HCL Package

**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.)

**Notes**

`collect-generation-2` is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations, where you can use `set-blocking-gen-num` instead.

**See also**

- `clean-generation-0`
- `collect-highest-generation`
- `expand-generation-1`
- `set-blocking-gen-num`
- `set-promotion-count`
  
  “Guidance for control of the storage management system” on page 112
  
  “Memory Management in 32-bit LispWorks” on page 117

---

**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 117

*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**
```lisp
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.
- `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

Copy-code-coverage-data
Copy-current-code-coverage
Load-code-coverage-data
Save-code-coverage-data
Save-current-code-coverage

Functions

Summary
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
The HCL Package

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-ccd 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 \texttt{set-array-weak} for a description of weak vectors.

See also \texttt{make-array} \linebreak \texttt{set-array-weak} \linebreak “Freeing of objects by the GC” on page 135

\textbf{create-macos-application-bundle} \hspace{1cm} \textit{Function}

\textbf{Summary} Creates a Mac OS X application bundle for the running LispWorks image.

\textbf{Package} hcl


\textbf{Arguments} \begin{itemize}
  \item \texttt{target-path} A pathname designator.
  \item \texttt{template-bundle} A pathname designator.
  \item \texttt{bundle-name} A string.
  \item \texttt{signature} A string.
  \item \texttt{package-type} A string.
  \item \texttt{extension} A string.
  \item \texttt{application-icns} A pathname designator.
  \item \texttt{identifier} A string.
  \item \texttt{version} A string.
  \item \texttt{build} A string.
  \item \texttt{version-string} A string.
  \item \texttt{help-book-folder} A string.
  \item \texttt{help-book-name} A string.
\end{itemize}
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 "LispWorks.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.
See also  

**create-temp-file**

**open-temp-file**

*Functions*

**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


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-stack-length

Function

Summary  Returns the size of the current stack.

Package  hcl
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, current-stack-length was not implemented. This is fixed in LispWorks 5.0 and later.

Example
(current-stack-length) => 16000

See also
extend-current-stack
*sg-default-size*

---

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

```lisp
(declaim (declaration ...))
```

There are currently no other supported values for `decl-name`.

**Notes**

declaration-information is part of the environment access API which is based on that specified in Common Lisp: the Language (2nd Edition).

**See also**

augment-environment

function-information

map-environment

variable-information

**default-package-use-list**

*Variable*

**Summary**

List of packages that newly created packages use by default.

**Package**

`hcl`

**Initial value**

`("CL" "LW" "HCL")`

**Description**

This variable is the default value of the :use keyword to defpackage, which specifies which existing packages the package being defined inherits from.
**default-profiler-collapse**  

*Variable*

**Summary**
Controls collapsing of the profile tree.

**Package**  
hcl

**Initial value**  
nil

**Description**
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*

**Summary**
The minimum percentage that the profiler will display in the output tree.

**Package**  
hcl

**Initial value**  
0

**Description**
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**

Summary: 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**

*Function*

**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.
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`

---

**delete-advice**

*Macro*

**Summary**

Removes a piece of advice.

**Package**

`hcl`

**Signature**

```lisp
(delete-advice dspec name => nil)
```

*dspec ::= fn-name | macro-name | (clos::method generic-fn-name ((class*)))*

**Arguments**

`dspec` Specifies the functional definition to which the piece of advice belongs. The specification contains the name of the associated function. In the case of a method the list of classes is used to identify from which particular method the advice should come. This list must correspond exactly with the classes corresponding to the specialized parameters for some method belonging to the generic function.

`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`.
**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**

---

### delivered-image-p

*Function*

**Summary**  
The predicate for whether the running image is a delivered image.

**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.

Description 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.lisp-works.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 `libLispWorks.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 `Android-Manifest.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-subdirectory `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 \textit{studio-p} is \texttt{t} it is "jniLibs/armeabi-v7a".
(ii) Otherwise if \textit{using-ndk} is \texttt{nil} it is "libs/armeabi-v7a".
(iii) Otherwise it is "jni".

2. If \textit{no-sub-dir} is non-nil, then the dynamic library is put in the \texttt{project-path} directory. If \textit{no-sub-dir} is \texttt{t}, the heap is put in the same place. Otherwise, the directory specified by \textit{no-sub-dir} is merged with the \texttt{project-path} to create the path where the heap goes.

Note: If \textit{no-sub-dir} is a pathname or string it must specify a directory, which can be an absolute path.

After \texttt{deliver-to-android-project} determines the names of the files and where they go, it calls \texttt{deliver}, passing \texttt{function}, the appropriate path, \texttt{level}, the Deliver keywords \texttt{:split}, \texttt{:exe-file}, \texttt{:dll-exports} and \texttt{:image-type} with the correct values for Android, and all the keyword arguments it was supplied except \texttt{library-name}, \texttt{using-ndk}, \texttt{studio-p} and \texttt{no-sub-dir}. The keywords that \texttt{deliver-to-android-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 \texttt{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 \texttt{using-ndk}. You need it when you use \texttt{ndk-build} to build dynamic libraries in your project. In Eclipse that is normally done by the CDT builder. You need \texttt{using-ndk} because the \texttt{ndk-build} removes all dynamic libraries from the \texttt{libs} sub-directory and its sub-directories, including the LispWorks dynamic library. The solution for this is to pass a true value for \texttt{using-ndk} to \texttt{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-7-0-0-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-7-0-0-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-direc-
tories (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**

**Summary**
Controls tracing.

**Package**
hcl

**Initial value**
nil

**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.

Description  
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.
- **show-counters**
  A boolean.
- **color-covered**
  A boolean, controlled by preferences.
- **color-uncovered**
  A boolean, controlled by preferences.
- **font-lock-p**
  `nil`, `t` or the keyword `:force`. The default value of `font-lock-p` is `t`.
- **comment-counters**
  A boolean, controlled by preferences.
- **real-filename**
  A pathname.
- **runtime-only**
  A boolean, controlled by preferences.

**Values**

- **result**
  An editor buffer object, or a list.

**Description**
The function `editor-color-code-coverage` displays code coverage in a LispWorks IDE Editor tool 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.

comment-counters controls 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 \textit{comment-counters} can be set in the \textbf{Preferences}... dialog for Code Coverage Browser, tab \textbf{Coloring}. The initial default value of \textit{comment-counters} is \texttt{t}.

\textit{runtime-only} controls whether to display only runtime forms, which means exclude forms that execute only at compile time or load time. The default for \textit{runtime-only} can be set in the \textbf{Preferences}... dialog for Code Coverage Browser, tab \textbf{Coloring}. The initial default value of \textit{runtime-only} is \texttt{nil}.

\textit{real-filename} may be used to specify the actual file to load. When it is non-nil, \textit{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 \texttt{code-coverage-data} object, while \textit{real-filename} is used as the actual text to load. Note that while \textit{filename} needs to be the same as the truename that the compiler used, it is not necessarily a real truename on the current machine.

\texttt{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 \texttt{editor-color-code-coverage} succeeds and \texttt{for-editing} is \texttt{nil}, it remembers that it generated the buffer for the \textit{filename}, and if it is called again with the same \textit{filename} and \texttt{for-editing} \texttt{nil} and succeeds, deletes the previous buffer.

See “Understanding the code coverage output” on page 106 for details of how to interpret the coloring.

\textbf{Notes}

\textit{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 Cover-
age browser uses it. Figuring out the truename on a different machine is not always easy. The best way is way is to use the one from the data, which you can find from a code-coverage-file-stats object if you already have it.

See also “Understanding the code coverage output” on page 106 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

- gen-num A generation number.
- size The amount (in bytes) by which the generation is to be enlarged.

Values result A boolean.

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 117

enlarge-static

Function

Summary
Enlarges the size of the first static segment in 32-bit LispWorks.

Package
hcl

Signature
enlarge-static size => result

Arguments
size A non-negative 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 t if the static segment was successfully enlarged, and nil otherwise.

Use room, with argument t, to find the size of the static segments, and thus the size by which to enlarge the first static segment.
Notes

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

in-static-area
room
set-default-segment-size
switch-static-allocation
“Memory Management in 32-bit LispWorks” on page 117

ensure-hash-entry

Function

Summary

Gets a value from a hash-table, adding a new value if this fails, all with the table locked.

Package

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 235

**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.
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

```
(if (check-something)
   (ok-code)
   (error-situation-forms (call-error)))
```

See also

- `without-code-coverage`
- `generate-code-coverage`
- `with-code-coverage-generation`

### expand-generation-1

**Function**

**Summary**
Controls expansion of generation 1 in 32-bit LispWorks.

**Package**
`hcl`

**Signature**
`expand-generation-1 on`

**Arguments**
`on`  
`t`, `nil` or 1.

**Description**
The function `expand-generation-1` controls the subsequent behavior of the garbage collector when insufficient space is freed by a `mark-and-sweep`. When this occurs, either generation 1 is expanded, or the objects in it are promoted.

If `on` is `nil`, generation 1 is never expanded.

If `on` is `t`, generation 1 is always expanded (rather than promotion) when needed.

If `on` is 1, generation 1 is only expanded if its current size is less than 500000 bytes. This is the initial setting.

**Notes**
`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 `set-default-segment-size`.

See also
- `clean-generation-0`
- `collect-generation-2`
- `collect-highest-generation`
- `mark-and-sweep`
- `set-default-segment-size`
- `set-gc-parameters`

“Memory Management in 32-bit LispWorks” on page 117

**extend-current-stack**

*Function*

**Summary**

Extends the current stack.

**Package**

hcl

**Signature**

`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:

```lisp
(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. 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
0.0
```
This example illustrates output in 64-bit LispWorks:

```lisp
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

<table>
<thead>
<tr>
<th>total</th>
<th>user</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td></td>
</tr>
<tr>
<td>total gc activity = 2.168062 / 2.126444</td>
<td></td>
</tr>
<tr>
<td>system</td>
<td>0.041618</td>
</tr>
<tr>
<td>Standard 0 (28545 calls) = 2.153866 / 2.119799</td>
<td></td>
</tr>
<tr>
<td>Standard 1 ( 1 calls) = 0.014176 / 0.006645</td>
<td></td>
</tr>
<tr>
<td>10006387712</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

See also start-gc-timing
time
“Guidance for control of the storage management system” on page 112

```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

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:

```lisp
(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 403

---

**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).
An external format specification, default value :default.

Values

- **string**: A string containing characters from `file`.

Description

Returns the entire contents of `file` (if `length` is `nil`), or the first `length` characters, as a string.

Example

```
CL-USER 26 > file-string "configure.lisp" :length 18
;; -*- Mode: Lisp;
```

See also

- `guess-external-format`

---

**file-writable-p**

*Function*

Summary

Tests whether a file is writable.

Package

`hcl`

Signature

`file-writable-p file => result`

Arguments

- **file**: A pathname, string or file-stream, designating a file.

Values

- **result**: `t` or `nil`

Description

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.

Example

```
CL-USER 44 > file-writable-p (sys:lispworks-file "private-patches/load.lisp")
T
```
**filter-code-coverage-data**  

*Function*

**Summary**
Filters information from a `code-coverage-data` object.

**Package**
hcl

**Signature**
```
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".
find-object-size

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 storage currently used to represent the object. If the object takes up no heap storage (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 storage.

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

\begin{verbatim}
(make-string 1000 :initial-element #\A
    :element-type 'base-char))
\end{verbatim}

See also

room

total-allocation

\section*{find-throw-tag \hspace{1cm} Function}

\textbf{Summary}

The predicate for whether there is a specific catch in the dynamic scope.

\textbf{Package}

hcl

\textbf{Signature}

\texttt{find-throw-tag tag }\Rightarrow\texttt{ result }

\textbf{Arguments}

tag \hspace{1cm} A catch tag.

\textbf{Values}

result \hspace{1cm} A boolean.

\textbf{Description}

The function \texttt{find-throw-tag} is the predicate for whether there is a catch in the dynamic scope with the supplied catch tag \texttt{tag}, so that \texttt{cl:throw} will succeed to throw to it.

\textbf{Notes}

\texttt{find-throw-tag} needs to traverse all the catch frames on the stack until it finds the tag, and therefore would be slower than 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 \texttt{find-throw-tag} is better because it eliminates the overhead of binding the special.

See also

\texttt{throw-if-tag-found}
The HCL Package

### finish-heavy-allocation

**Function**

**Summary**
Tells the system that allocation of many long-lived objects is over.

**Package**
hcl

**Signature**
finish-heavy-allocation

**Description**
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-free-action

\textbf{function-information} \quad \textit{Function}

Summary
Return information about the function bindings of a symbol in an environment.

Package
hcl

Signature
\texttt{function-information function-name} \&optional \texttt{env} \Rightarrow \texttt{kind, localp, decls}

Arguments
\texttt{function-name} \quad A function name
\texttt{env} \quad An environment or \texttt{nil}

Values
\texttt{kind} \quad Either \texttt{nil}, or one of the keywords \texttt{:special, :lexical, :symbol-macro} and \texttt{:constant}.
\texttt{localp} \quad A boolean
\texttt{decls} \quad An a-list

Description
The function \texttt{function-information} returns information about how \texttt{function-name} is bound in the environment \texttt{env}. \texttt{function-name} can be a symbol or \texttt{setf} function name.

The value of \texttt{kind} will be as follows:

\begin{itemize}
  \item \texttt{nil} \quad There is no information about \texttt{variable} in \texttt{env}
  \item \texttt{:macro} \quad \texttt{function-name} has a macro binding in \texttt{env}
\end{itemize}
: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 flet) or false if function-name has global scope (for example defun).

decls is an a-list of declarations that refer to function-name.
The cdr of each pair is specified according to the car of the pair as follows:

dynamic-extent
The cdr is non-nil if function-name is declared dynamic-extent in env.

inline
The cdr is inline or notinline if function-name is explicitly declared inline or notinline in env. The cdr is nil (or the pair is omitted) if this information is not known.

ftype
The cdr is the type specifier that is declared for function-name in 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
augment-environment
declaration-information
map-environment
variable-information
The HCL Package

**get-code-coverage-delta**

**reset-code-coverage-snapshot**

**set-code-coverage-snapshot**

**Functions**

**Summary**
Generate "deltas", which are **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 memory.

The function **set-code-coverage-snapshot** creates a snapshot 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-coverage and restore-code-coverage-data also eliminate 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.
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
The HCL Package

**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.

**Compatibility notes**

In 32-bit LispWorks, **gc-generation** simply calls **mark-and-sweep**. This has a similar effect, but two significant differences must be noted:

1. by default, **gc-generation** promotes the young generations, so repeated calls to **gc-generation** will promote everything to generation `gen-num` or generation `block` (whichever is lower). In contrast **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

```
(gc-generation t)
```

is intended as the replacement of

```
(mark-and-sweep 2)
```

**See also**

- **clean-down**
- **mark-and-sweep**
- **marking-gc**
- **set-blocking-gen-num**
- “Guidance for control of the storage management system” on page 112

**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 117

generate-code-coverage  Function
Summary  Switches code coverage generation on or off.
Package  hcl
Signature  generate-code-coverage &key on atomic-p counters force count-
imPLICIT-branch => on
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>A boolean.</td>
</tr>
<tr>
<td>atomic-p</td>
<td>A boolean.</td>
</tr>
<tr>
<td>counters</td>
<td>A boolean.</td>
</tr>
<tr>
<td>force</td>
<td>A boolean.</td>
</tr>
<tr>
<td>count-implicit-branch</td>
<td>A boolean.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>A boolean.</td>
</tr>
</tbody>
</table>

Description

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 `counters` is `t`.
force, if true, forces generating counters in code that is marked not to generate counters by without-code-coverage or error-situation-forms. The default value of force is nil.

count-implicit-branch controls whether to generate counters for implicit branches. Implicit branches are generated by macros like cl:when, where the source only contains the "then" branch, and the "else" branch (which returns nil) is implicit. The other macros are cl:unless (when it is an implicit "then"), and the switch macros cl:cond, cl:case and cl:tpycase when they do not have a t or otherwise clause. When 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 cl:when this is the number of times that the condition returned nil; for 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 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 hidden-partial form (see “Understanding the code coverage output” on page 106).

The default value of count-implicit-branch is t.

Notes

If generate-code-coverage is called outside the body of with-code-coverage-generation, it switches the generation globally. Inside the body of with-code-coverage-generation it switches the generation within the scope of the surrounding with-code-coverage-generation, but has no effect once this with-code-coverage-generation exited.

See also

code-coverage-data-generate-coloring-html
editor-color-code-coverage
error-situation-forms
with-code-coverage-generation
**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 117

**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 > (print (get-gc-parameters t))
((:ENLARGE-BY-SEGMENTS . 10)
 (:MINIMUM-FOR-PROMOTE . 10000)
 (: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 117
**get-temp-directory**  
*Function*

<table>
<thead>
<tr>
<th>Summary</th>
<th>Returns a directory that can be used for temporary files.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>hcl</td>
</tr>
<tr>
<td>Signature</td>
<td>get-temp-directory =&gt; directory</td>
</tr>
<tr>
<td>Values</td>
<td>directory</td>
</tr>
<tr>
<td>Description</td>
<td>The function get-temp-directory returns a directory which is likely to be writable and can be used for temporary files.</td>
</tr>
<tr>
<td>Notes</td>
<td>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.</td>
</tr>
</tbody>
</table>
| See also | create-temp-file
          | example-compile-file
          | open-temp-file |

**get-working-directory**  
*Function*

<table>
<thead>
<tr>
<th>Summary</th>
<th>Finds the current working directory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>hcl</td>
</tr>
<tr>
<td>Signature</td>
<td>get-working-directory =&gt; cwd</td>
</tr>
<tr>
<td>Arguments</td>
<td>None.</td>
</tr>
<tr>
<td>Values</td>
<td>cwd</td>
</tr>
</tbody>
</table>
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

Function

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`.

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 235
**handle-existing-defpackage**

**Variable**

**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

**Initial value**

`:warn`

**Description**

The variable `*handle-old-in-package*` controls what happens when a CLtL1-style `in-package` form is processed. This refers to the specification in Common Lisp the Language, first Edition, which preceded ANSI Common Lisp and specified `in-package` as a function with keyword arguments.

The allowed values are as follows:

- **`:quiet`**
  
  Quietly use the CLtL1 definition of the `in-package` function.

- **`:warn`**
  
  Signal a warning and use the old definition.

- **`:error`**
  
  Signal a continuable error.

**See also**

- `*handle-old-in-package-used-as-make-package*`
**Variable**

*handle-old-in-package-used-as-make-package*

**Summary**
Controls the handling of CLtL1-style `in-package` forms.

**Package**
hcl

**Initial value**
:quiet

**Description**
The variable *handle-old-in-package-used-as-make-package* controls what happens when a CLtL1-style `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 `in-package` as a function with keyword arguments.

The allowed values are as follows:

:quiet Handle according to the value of *handle-old-in-package*.

:warn Signal a warning and create the package.

:error Signal a continuable error.

See also *handle-old-in-package*

---

**Function**

hash-table-weak-kind

**Summary**
Returns the weak kind of a hash table.

**Signature**

`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  `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`. 

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.

Note: callback works only when the file was generated by LispWorks 7.0 or later.

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 `eval` is `t` to give the same behavior as in LispWorks 6.1 and earlier versions. Passing `eval` as `nil` and using a callback is probably a better way of transferring data around, because it avoids the calls to `eval`. If needed, `callback` can call `eval` explicitly.

2. All x86/x64 and ARM architectures have the same byte order, so `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 Mac OS X) and SPARC (old Solaris).
3. **load-data-file** returns the same value as **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 **callback** argument is used to perform any required side effects.

4. **load-data-file** does not do any read operation, but if the forms in the file contain symbols (except **nil**) such symbols need to be interned.

Compatibility notes
1. In LispWorks 6.1 and earlier versions **load-data-file** was in the SYSTEM package. It is still exported from SYSTEM for backwards compatibility.

2. In LispWorks 6.1 and earlier versions **load-data-file** gave errors if the type was not recognized, but now by default it allows any type.

3. **callback** works only when the fasl file was generated by LispWorks 7.0 or later.

Examples
For a simple example see **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 414

*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  

Function

Summary Creates a "ring" object.

Package hcl

Signature make-ring size name &optional delete-function => ring

Arguments size A positive fixnum.
<table>
<thead>
<tr>
<th>name</th>
<th>A string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete-function</td>
<td>A function designator for a function of one argument.</td>
</tr>
</tbody>
</table>

**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 `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 `ring-pop` removes from the ring the element before the insertion point, and returns that element. Thus when using `ring-push` and `ring-pop` on their own, the ring behaves like a stack with limited length.

`rotate-ring` can be used to move the insertion point. `ring-ref` can be used to index into the ring. `map-ring`, `position-in-ring`, and `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}\linebreak \texttt{ring-pop}\linebreak \texttt{rotate-ring}\linebreak \texttt{ring-ref}\linebreak \texttt{ring-length}\linebreak \texttt{ringp}\linebreak \texttt{ring-name}\linebreak \texttt{map-ring}\linebreak \texttt{position-in-ring}\linebreak \texttt{with-ring-locked}

\begin{description}
\item[\textbf{make-unlocked-queue}]\linebreak \textbf{Summary} Create and use an \texttt{unlocked-queue} object.
\item[\textbf{Package}] \texttt{hcl}
\item[\textbf{Signature}] \texttt{make-unlocked-queue &key size name => unlocked-queue}\linebreak \texttt{unlocked-queue-read unlocked-queue => object}\linebreak \texttt{unlocked-queue-ready unlocked-queue => result}\linebreak \texttt{unlocked-queue-send unlocked-queue object => object}\n\item[\textbf{Arguments}] \begin{itemize}
\item \texttt{size} A positive integer.
\item \texttt{name} A Lisp object.
\item \texttt{unlocked-queue} An \texttt{unlocked-queue} object.
\item \texttt{object} A Lisp object.
\end{itemize}
\end{description}
Values

object  A Lisp object.
result  A boolean.

Description

The function make-unlocked-queue creates a new, empty unlocked-queue object.

The functions unlocked-queue-read, unlocked-queue-ready and unlocked-queue-send use an 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. size 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-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.

See also

make-mailbox
unlocked-queue

map-ring  

Function

Summary

Calls a function on each element of a ring, modifying the element.

Package

hcl

Signature

map-ring ring function
The HCL Package

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

**Function**

Summary

Garbage collects a specified generation in 32-bit LispWorks. This function is deprecated: use `gc-generation` instead.

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 storage (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 storage management system” on page 112
### `*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**

```lisp
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

USER 11 > (sum 3 0)
0 SUM > (3 0)
  1 SUM > (2 0)
   2 SUM > (1 0)
    3 SUM > (0 0)
     3 SUM < (0)
      2 SUM < (1)
       1 SUM < (3)
        0 SUM < (6)
         6
```

**Notes**  
`*max-trace-indent*` is an extension to Common Lisp.

**See also**  
trace
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.

Merging means taking all the files from ccd1 together with those files from ccd2 which do not have information in ccd1. For files that appear in both ccd1 and ccd2, the information in ccd2 is ignored.

merge-code-coverage-data creates a new code-coverage-data object containing the information for each file, and with name name.

destructive-merge-code-coverage-data adds to ccd1 those files from ccd2 which are not already there, and returns ccd1.

If either of the datas is the internal 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”
code-coverage-data

modify-hash

Summary
Reads and writes an entry in a hash table atomically.

Package
hcl

Signature
modify-hash hash-table key function => new-value, key

Arguments
hash-table A hash table.
key An object.
function A function designator.

Values
new-value An object.
key An object.

Description
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:

(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.

Notes

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
with-hash-table-locked
"Atomicity and thread-safety of the LispWorks implementation" on page 229
"Modifying a hash table with multiprocessing" on page 235

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 117

*packages-for-warn-on-redefinition*
Variable

Summary
A list specifying packages whose symbols should be checked on attempted definitions.

Package
hcl

Initial value
(:implementation)

Description
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:def-parameter 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 rebind- ing *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 is 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 70

**parse-float**

Function

Summary

Parses a float from a string and returns it as float.

Package

hcl
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*. 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
```
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 position-in-ring finds in the ring ring the first element that matches item and returns its index, or nil if there is no match. The search starts from index 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 ring-ref with indices index, index+1, ..., length-1. It does not wrap around, so elements between indices 0 and index are not tested.

                 The function position-in-ring-forward does the same except that it searches from index "forward" to the insertion point. In other words, it tests the elements that would be returned by ring-ref with indices index, index-1, ..., 0.

                 The comparison is done by calling test, with item as first argument and each element in the ring as the second argument. The default value of test is eql.

Notes            test is called with the ring locked.

Compatibility    In LispWorks 6.1 and earlier versions, these functions are called find-in-ring and find-in-ring-forward. They have been renamed to match the Common Lisp convention that a function returning an index is named position-*. The
old names are retained for backwards compatibility, but are deprecated.

See also
make-ring
map-ring

### print-profile-list

**Function**

**Summary**
Prints a report of symbols that have been profiled.

**Package**
hcl

**Signature**
print-profile-list &key sort limit cutoff collapse => nil

**Arguments**
- **sort**: `:call`, `:profile` or `:top`
- **limit**: An integer.
- **collapse**: A generalized boolean.
- **cutoff**: A real number.

**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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>called</th>
<th>profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>top</td>
<td>(%)</td>
</tr>
<tr>
<td>CADR</td>
<td>5000012</td>
<td>13</td>
</tr>
<tr>
<td>(  4)</td>
<td>13 (  4)</td>
<td></td>
</tr>
<tr>
<td>CDDR</td>
<td>3000000</td>
<td>3</td>
</tr>
<tr>
<td>(  1)</td>
<td>3 (  1)</td>
<td></td>
</tr>
<tr>
<td>EQL</td>
<td>2000202</td>
<td>4</td>
</tr>
<tr>
<td>(  1)</td>
<td>4 (  1)</td>
<td></td>
</tr>
<tr>
<td>FIXNUMP</td>
<td>2000003</td>
<td>2</td>
</tr>
<tr>
<td>(  1)</td>
<td>2 (  1)</td>
<td></td>
</tr>
<tr>
<td>CAR</td>
<td>1000000</td>
<td>1</td>
</tr>
<tr>
<td>(  0)</td>
<td>1 (  0)</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1000000</td>
<td>3</td>
</tr>
<tr>
<td>(  1)</td>
<td>3 (  1)</td>
<td></td>
</tr>
<tr>
<td>CAADR</td>
<td>1000000</td>
<td>2</td>
</tr>
<tr>
<td>(  1)</td>
<td>2 (  1)</td>
<td></td>
</tr>
<tr>
<td>1+</td>
<td>1000000</td>
<td>2</td>
</tr>
<tr>
<td>(  1)</td>
<td>2 (  1)</td>
<td></td>
</tr>
</tbody>
</table>

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

Runs the specified forms, and prints a performance profile.

Package

hcl

Signature

`profile &body forms => final`

Arguments

`forms` The forms making up the program being profiled.

Values

`final` The result of evaluating the final form.
Description

This macro starts up the LispWorks program profiler. This tool is useful for determining the time critical elements of a program.

At a regular time interval the Lisp process is halted and the execution stack is scanned for the presence of any symbols in the list *profile-symbol-list*. Counters are maintained for the number of calls to each symbol, the total number of times the symbol is found on the stack, and the number of times the profiler finds the symbol on the top of the stack.

This information is then presented as absolute numbers and as a percentage of the total number of calls to the profiler. These figures taken together give useful information about which functions the program spends most of its time executing.
Examples

(set-up-profiler
  :symbols '(* sqrt floor))
=>
20

(profile
  (let ((x 1))
    (loop for a from 1 to 500000
          do (setq x (floor (* a (sqrt x))))
            finally (return (integer-length x)))))
=>
profile-stacks called 401 times

Call tree
Symbol                      seen (%)  
1: "CAPI Execution Listener 1"  401
(100)
2: *                             5 (  1)
2: SQRT                          4 (  1)
2: FLOOR                         1 (  0)

Cumulative profile summary
Symbol          called profile (%)  top (%) 
*               0      5 (  1)    0 (  0)
SQRT            0      4 (  1)    1 (  0)
FLOOR           0      1 (  0)    0 (  0)

On average 1.0 stacks profiled each profiler invocation
Top of stack not monitored 100% of the time

See also
  print-profile-list
  *profile-symbol-list*
  set-up-profiler

  “Guidance for control of the storage management system” on page 112

*profiler-threshold*     Variable

Summary     Controls which symbols are profiled on repeated profiling runs.
The HCL Package

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

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  

profiler-tree-from-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-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-tree-to-function

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
### The HCL Package

#### 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`

### 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**  
A generalized boolean.
Values  

\textit{new-size} \hspace{1cm} \text{A positive integer.}

Description  

The function \texttt{reduce-memory} frees memory and tries to reduce the size of the Lisp image, without enlarging it even temporarily. It is implemented only in 32-bit LispWorks.

\texttt{reduce-memory} has the same effect as \texttt{clean-down}, except that \texttt{clean-down} may temporarily increase the size of the image in order to be able to promote from lower generations. \texttt{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.

\texttt{reduce-memory} is intended to be used when the operating system signals that the memory is low. Using \texttt{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 \texttt{clean-down} should do a better job (and you might also consider \texttt{try-move-in-generation}).

If \texttt{full} is \texttt{nil}, \texttt{reduce-memory} frees memory and promotes live objects to generation 2. When \texttt{full} is non-nil, \texttt{reduce-memory} frees and promotes to generation 3. The default value of \texttt{full} is \texttt{nil}.

\texttt{reduce-memory} returns the new size of the Lisp image after reduction, in bytes.

Notes  

1. The default of \texttt{full} is \texttt{nil}, which is different from \texttt{clean-down} where it defaults to \texttt{t}.

2. \texttt{reduce-memory} with no argument or \texttt{nil} differs from \texttt{(clean-down nil)} by trying to reduce the memory. \texttt{(clean-down nil)} frees and promotes, but does not try to reduce the size (and may actually increase it).

3. \texttt{reduce-memory} is implemented only in 32-bit LispWorks.
See also

clean-down
try-move-in-generation

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

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

`remove-symbol-profiler` removes a symbol from *profile-symbol-list*, the list of profiled symbols.

See also

`add-symbol-profiler`
*profile-symbol-list*
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 *profile-symbol-list*. 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 profile-symbol-list 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*

**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*

**Summary**
Gets the element count and maximum size of a ring.
The HCL Package

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 (rotate-ring ring 1). The default value of remove is t.

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:

(values (ring-pop ring) (ring-ref ring 0))
(values (ring-pop ring t) (ring-ref ring 0))
(values (ring-ref ring 0) (rotate-ring ring 1))

See also: ring-push
make-ring
rotate-ring
ring-ref
ring-length

Function:

ring-push

Summary: Adds a Lisp object to a ring.
The HCL Package

Package    hcl
Signature   ring-push object ring => object
Arguments   object  A Lisp object.
             ring    A ring object created by make-ring.
Values      object  A Lisp object.
Description The function ring-push adds any Lisp object as an element of the ring before the "insertion position", which means that a following 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 insertion 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.
The HCL Package

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**

---

**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-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.
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 150.

`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 Mac OS X Dock) will run itself only if the default session is "LispWorks Release".

See also

- `save-image`
- `save-image-with-bundle`

### save-image

**Function**

**Summary**

Saves the image to a new file.

**Package**

hcl

**Signature**

```
save-image filename &key dll-exports dll-added-files automatic-init gc type normal-gc restart-function multiprocessing console environment remarks clean-down image-type split => nil
```
The console argument is available only in LispWorks for Windows and LispWorks for Macintosh.

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>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.</td>
</tr>
<tr>
<td>dll-exports</td>
<td>A list of strings, or the keyword :default.</td>
</tr>
<tr>
<td>dll-added-files</td>
<td>A list of strings.</td>
</tr>
<tr>
<td>automatic-init</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>gc</td>
<td>If non-nil, there is a garbage collection before the image is saved. The default value is t.</td>
</tr>
<tr>
<td>type</td>
<td>Determines if some global variables are cleared before the image is saved. You can generally use the default value, which is :user.</td>
</tr>
<tr>
<td>normal-gc</td>
<td>If this is t the function normal-gc is called before the image is saved. The default is t.</td>
</tr>
<tr>
<td>restart-function</td>
<td>A function to be called on restart.</td>
</tr>
<tr>
<td>multiprocessing</td>
<td>Controls whether multiprocessing is enabled on restart. Possible values are discussed below.</td>
</tr>
<tr>
<td>console</td>
<td>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.</td>
</tr>
<tr>
<td>environment</td>
<td>environment controls whether the LispWorks environment is started on restart. Possible values are discussed below.</td>
</tr>
</tbody>
</table>
The HCL Package

- **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` command 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/PowerPC, 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 auto-

matic 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 com-

piler (`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 Lisp-

Works 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 396.

- Calling **QuitLispWorks** when required.

- Changing calls that involve complex C structs or even

  C++ objects into plain calls, because accessing C struc-

  tures 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-func-

  callable**.

- Adding code that runs automatically inside the call to

  **dlopen**, by using `__attribute__ ((constructor))`
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 as an application bundle on the Macintosh (for example by using create-macos-application-bundle), split can be the symbol :resources. This places the Lisp heap file in the Resources directory of the bundle, rather than in the Contents/MacOS directory alongside the executable, which allows the heap to be included in the signature of the bundle.

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.
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.

customize is implemented only in LispWorks for Windows and LispWorks for Macintosh. The possible values for customize 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 customize 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 `c:/build-my-image.lisp`:

```lisp
(load-all-patches)
(load "my-code")
(save-image "my-image")
```

Then run LispWorks with the command line argument `-build c:/build-my-image.lisp` to save the image `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:

```lisp
(load-all-patches)
(load "my-code")
#:cocoa
(compile-file-if-needed
  (example-file
    "configuration/macros-application-bundle")
  :load t)
(save-image
 #:cocoa
 (write-macos-application-bundle
  "~/Applications/LispWorks 7.0/My LispWorks.app")
#:cocoa
 "my-lispworks")
```

See also

- `deliver`
- `dll-quit`
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 Lisp-Works 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.
The function `save-universal-from-script` is deprecated.

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

**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 generation is required to remove the pointer. Note that by default the system does not automatically GC the blocking generation 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
set-hash-table-weak
make-array
mark-and-sweep
“Freeing of objects by the GC” on page 135

set-default-generation

Function

Summary
Set the current generation for storage 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 storage 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
(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"
"expand-generation-1"
"get-default-generation"
"set-promotion-count"
"*symbol-alloc-gen-num*"
“Memory Management in 32-bit LispWorks” on page 117

**set-gc-parameters**

**Function**

**Summary**
Sets the parameters for the garbage collector in 32-bit LispWorks. This function is deprecated.

**Package**
hcl

**Signature**

`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, 10000000) 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 64KB. 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 117

---

**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 135

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen-num</td>
<td>The generation to be affected.</td>
</tr>
<tr>
<td>size</td>
<td>The size (in bytes) to set the segment to.</td>
</tr>
<tr>
<td>segment</td>
<td>An integer specifying the segment to be affected. The default value is 0, meaning the first segment of the generation.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generation-size</td>
<td>A list showing information for the generation just specified in the call.</td>
</tr>
</tbody>
</table>

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.
set-minimum-free-space is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations.

See also
- clean-generation-0
- collect-generation-2
- collect-highest-generation
- expand-generation-1
- room
- set-promotion-count

“Memory Management in 32-bit LispWorks” on page 117

**set-process-profiling**

*Function*

**Summary**
Controls the set of processes that are profiled.

**Package**
hcl

**Signature**

```
set-process-profiling flag processes
```

**Arguments**

- `processes`: One of :current, :all, a `mp:process` object, or a list of `mp:process` objects which may also contain :current.

**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.

A `mp:process` object

Means that process.

A list

Means the processes in that list. The list can contain the symbol `:current`, which is interpreted as described above.

`set-process-profiling` can be called whether or not the profiler is collecting information. See `start-profiling` and `stop-profiling`.

**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

**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
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`. 
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
### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbols</td>
<td>A symbol or a list of symbols.</td>
</tr>
<tr>
<td>packages</td>
<td>A valid package name, or a list of package names, or :all.</td>
</tr>
<tr>
<td>kind</td>
<td>:profile, :virtual or :real.</td>
</tr>
<tr>
<td>interval</td>
<td>An integer greater than or equal to 10000.</td>
</tr>
<tr>
<td>limit</td>
<td>An integer or nil.</td>
</tr>
<tr>
<td>cutoff</td>
<td>An integer or nil.</td>
</tr>
<tr>
<td>collapse</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>style</td>
<td>:tree, :list or nil.</td>
</tr>
<tr>
<td>gc</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>call-counter</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>show-unknown-frames</td>
<td>A generalized boolean.</td>
</tr>
</tbody>
</table>

### Values

The time interval is returned.

### Description

set-up-profiler is used to declare the values of the parameters of the profiling function. Three values are required, as follows.

- symbols, if non-nil, specifies which symbols are to be monitored by the profiler. Each symbol in symbols is checked to see if it is suitable for profiling and if so it is added to the list *profile-symbol-list*.

If symbols is not passed then packages specifies which symbols are to be monitored. If packages is :all, then all packages are monitored. All the symbols in the packages are checked as above. If a symbols argument is present then packages is ignored.

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`.

`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`.

`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` on Intel-based platforms and `t` on other platforms. This is because the counting significantly affects the performance of applications using Symmetric Multiprocessing (SMP).

Notes

1. 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.

2. `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.

Example

```
(set-up-profiler :symbols '(car cdr)
                 :interval 50000)
```

On Unix/Linux/Mac OS X:

```
(set-up-profiler :symbols '(car cdr)
                 :kind :profile :interval 50000)
```

See also

- `add-symbol-profiler`
- `*default-profiler-collapse*`
- `*default-profiler-cutoff*`
- `*default-profiler-limit*`
The HCL Package

profile
*profile-symbol-list*
remove-symbol-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

\textit{bool} \quad \text{If t, source level debugging is on.}

Description
Returns \textit{t} if source level debugging is on for compiled code; otherwise returns \textit{nil}.

See also \textit{toggle-source-debugging}

\textbf{start-gc-timing}
\textbf{stop-gc-timing}
\textbf{get-gc-timing}

Functions

Summary
Time Garbage Collector (GC) operations.

Package
\textit{hcl}

Signature
\textit{start-gc-timing} \&key \textit{initialize}
\textit{stop-gc-timing}
\textit{get-gc-timing} \&key \textit{reset}

Arguments
\textit{initialize} \quad A generalized boolean.
\textit{reset} \quad A generalized boolean.

Description
The functions \textit{start-gc-timing}, \textit{stop-gc-timing} and \textit{get-gc-timing} time Garbage Collector (GC) operations.

\textit{start-gc-timing} causes the system to start collecting GC timing. If \textit{initialize} is non-nil, \textit{start-gc-timing} also resets the Garbage Collector times to 0. The default value of \textit{initialize} is \textit{t}.

\textit{stop-gc-timing} stops collecting GC timing, but does not affect the times.

\textit{get-gc-timing} returns the GC timing as a plist of the form...
The HCL Package

::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

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.

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.

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.

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-profile control the behavior of the profiler. See the documentation for set-up-profile.
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:

```lisp
(start-profiling :processes :current)
(do-interesting-work)
```

Temporarily suspend profiling:

```lisp
(stop-profiling :print nil)
(do-uninteresting-work)
```

Resume profiling:

```lisp
(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 storage management system” on page 112

**stop-profiling**  

**Function**

**Summary**

Stops collecting profiling information.

**Package**

`hcl`

**Signature**

`stop-profiling &key print stream`

**Arguments**

- `print` A generalized boolean.
- `stream` An output stream.
- `suspend` A generalized boolean.
Description

The function **stop-profiling** stops collecting profiling information, and optionally prints the results.

If **suspend** is false, then the next call to **start-profiling** must pass **initialize t** or omit the **initialize** argument. In addition, if **print** is true, then the collected profiler information is printed.

If **suspend** is true, then the profiler is put into a suspended state where no profiling information is collected, but can be restarted by calling

```lisp
(start-profiling :initialize nil)
```

The default value of **print** is **t** and the default value of **suspend** is (not **print**). The value of **print** is ignored if **suspend** is true.

**stream** specifies the stream for output when **print** is non-nil. It is ignored when **print** is **nil**. The default value of **stream** is the value of ***trace-output***.

Notes

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 storage management system” on page 112

---

**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`
### The HCL Package

**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 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.

**See also**
- `sweep-gen-num-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  

\texttt{switch-static-allocation} returns the previous setting of \texttt{flag}.

Description  

Objects in the static area are garbage-collected, but not moved.

You should avoid using this function.

See also  

\texttt{enlarge-static}  
\texttt{in-static-area}

\*\texttt{symbol.alloc-gen-num}\*  

\textit{Variable}

Summary  

Specifies the generation in which interned symbols and their symbol names are allocated.

Package  

\texttt{hcl}

Initial value  

2 in 32-bit LispWorks, 3 in 64-bit LispWorks

See also  

\texttt{allocation-in-gen-num}  
\texttt{get-default-generation}  
\texttt{set-default-generation}  

“Allocation of interned symbols and packages” on page 133

\texttt{symbol-dynamically-bound-p}  

\textit{Function}

Summary  

The predicate for whether a symbol is dynamically bound.

Package  

\texttt{hcl}

Signature  

\texttt{symbol-dynamically-bound-p symbol => result}

Arguments  

\texttt{symbol}  

A non-nil symbol.
<table>
<thead>
<tr>
<th>Values</th>
<th>result</th>
<th>A boolean.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The function <code>symbol-dynamically-bound-p</code> is the predicate for whether the symbol <code>symbol</code> is dynamically bound in the current environment.</td>
<td></td>
</tr>
<tr>
<td>See also</td>
<td>Chapter 28, “Miscellaneous Utilities”</td>
<td></td>
</tr>
</tbody>
</table>

### throw-if-tag-found

**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**
toggle-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.
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**

Calculate memory consumed since the image was started.

**Summary**

**Package** hcl

**Signature** total-allocation

**Arguments** None.

**Values** Returns the amount allocated
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 117

*traced-arglist*

**Variable**

The list of arguments given to the function being traced.

**Summary**

**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:

```
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**
*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**
*trace-indent-width* is an extension to Common Lisp.
The current depth of tracing.

Package  hcl
Initial value  0

Description  *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 structures 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 independently of the usual printing mechanism, which is governed by *print-circle*. *print-circle* is bound to the value of *trace-print-circle* while printing tracing information.

Example  
USER 19 > (setq *trace-print-circle* t)
T
USER 20 > (defun circ (l)
    (rplacd (last l) 1)
  1)
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

Notes  
*trace-print-circle* is an extension to Common Lisp.

See also  
trace

**trace-print-length**  
*Variable*  

Summary  
The number of components of an object that are printed in trace output.
The HCL Package

Package  hcl
Initial value  100

Description  *
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 *
trace-print-length*
components are printed.

*print-length*
is bound to the value of *
trace-print-length*
while printing tracing information. If *
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. \texttt{*print-pretty*} is bound to the value of \texttt{*trace-print-pretty*} while printing tracing information.

**Examples**

```lisp
USER 6> (trace macroexpand-1)
MACROEXPAND-1
USER 7> (setq *trace-print-pretty* t
\texttt{*print-pretty* nil})
NIL
USER 8> (defmacro sum (n)
\texttt{'(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**

\texttt{*trace-print-pretty*} is an extension to Common Lisp.

**See also**

\texttt{trace}

---

\textbf{*trace-verbose*}  

**Variable**

**Summary**

Controls how arguments and values are printed in trace output.

**Package**

\texttt{hcl}

**Initial value**

\texttt{: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.

See also  trace

try-compact-in-generation  

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.
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 126.

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 117
**try-move-in-generation**

*Function*

**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 `t` if at least one segment was moved 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-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) \texttt{try-move-in-generation} moves only moderately fragmented segments.

If \textit{time-threshold} is negative, then \texttt{try-move-in-generation} does not actually move any segments. \textit{Result} is a boolean indicating whether \texttt{try-move-in-generation} would actually try to move a segment if it were to be called with a positive \textit{time-threshold} and the other arguments unchanged.

This function is typically used after a call to \texttt{check-fragmentation}. For more information, see “Controlling Fragmentation” on page 126.

Notes \texttt{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 \texttt{marking-gc} with the \textit{what-to-copy} argument offers similar functionality (although \texttt{set-blocking-gen-num} is intended to solve the problem of fragmentation automatically).

See also \texttt{check-fragmentation} \texttt{try-compact-in-generation} “Guidance for control of the storage management system” on page 112

\textbf{unlocked-queue} \hspace{1cm} \textit{Type}

\textbf{Summary} A fast queue.

\textbf{Package} \texttt{hcl}

\textbf{Description} The type \texttt{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 \texttt{mp: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.

Description  
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

(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**: variable has a lexical binding in env
- **:symbol-macro**: variable has a symbol-macro binding in env
- **:constant**: variable has a constant binding in env

localp will be true if variable is bound by a form that has lexical scope (for example let, lambda) or false if variable has global scope (for example defvar).

decls is an a-list of declarations that refer to variable. The cdr of each pair is specified according to the car of the pair as follows:

- **dynamic-extent**: The cdr is non-nil if variable is declared dynamic-extent in env.
- **ignore**: The cdr is non-nil if variable is declared ignore in env.
type

The cdr is the type specifier that is declared for variable in env if any.

Notes

variable-information is part of the environment access API which is based on that specified in Common Lisp: the Language (2nd Edition).

See also

augment-environment
declaration-information
function-information
map-environment

who-binds

Function

Summary

Returns the definitions which bind a special variable.

Package

hcl

Signature

\texttt{who-binds} \texttt{symbol} \Rightarrow \texttt{result}

Arguments

\texttt{symbol} \quad \text{A special variable.}

Values

\texttt{result} \quad \text{A list.}

Description

The function \texttt{who-binds} returns a list of dspecs naming the definitions which bind the special variable \texttt{symbol}.

Notes

The cross-referencing information used by \texttt{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 dspeca => 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.
### who-references

**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.

**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 A boolean.

Values
result The result of executing body.

Description
The macro with-code-coverage-generation switches code coverage generation on or off inside the dynamic scope of body.

The arguments are interpreted as by generate-code-coverage.

See also
generate-code-coverage

with-ensuring-gethash

Macro

Summary
A thread-safe way to get a value from a hash-table, adding a value if the key is not present. Allows a complicated form to construct the new value.

Package
hcl
Signature

\texttt{with-ensuring-gethash} \texttt{key hash-table \&key constructor constructor-form in-lock-constructor in-lock-constructor-form \=> result}

Arguments

- \texttt{key} A Lisp object.
- \texttt{hash-table} A hash-table.
- \texttt{constructor} A function designator for a function of no arguments, or \texttt{nil}.
- \texttt{constructor-form} A Lisp form, or \texttt{nil}.
- \texttt{in-lock-constructor} A function designator for a function of one argument, or \texttt{nil}.
- \texttt{in-lock-constructor-form} A Lisp form, or \texttt{nil}.

Values

- \texttt{result} A Lisp object.

Description

The macro \texttt{with-ensuring-gethash} gets the value for the key \texttt{key} from the hash table \texttt{hash-table}, and if this fails constructs a new value, puts it in the table and returns it. \texttt{with-ensuring-gethash} does this in a thread-safe way, which means that all threads calling it with the same \texttt{key} and \texttt{hash-table} return the same value (as long as nothing removes it from the table).

Only one of \texttt{constructor-form} or \texttt{constructor} can be non-nil. When \texttt{key} is not found, \texttt{constructor-form} or \texttt{constructor} is used to construct the new value. If \texttt{constructor} is non-nil, it is called without arguments. If \texttt{constructor-form} is non-nil, it is executed. If both are \texttt{nil}, the new value is \texttt{nil} unless the \texttt{in-lock-constructor} or \texttt{in-lock-constructor-form} construct it. The call or execution of the \texttt{constructor} or \texttt{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 235

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
with-hash-table-locked

Macro

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 229

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`.
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 117

### 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 (fasl-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

- fasl-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.
- 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 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 \texttt{delete-on-error} is \texttt{nil} then the file is left as it is. The default value of \texttt{delete-on-error} is \texttt{t}.

\texttt{dump-standard-objects} specifies what to do when trying to dump a standard object (that is, an instance of a subclass of \texttt{standard-object}) which does not have a user-defined \texttt{make-load-form}. If \texttt{dump-standard-objects} is \texttt{nil}, an error is signaled. If \texttt{dump-standard-objects} is non-nil, the instance is dumped using \texttt{make-load-form-saving-slots}. The default value of \texttt{dump-standard-objects} is \texttt{nil}.

When the generated file is loaded by \texttt{load-data-file}, the forms are loaded and by default evaluated, though \texttt{load-data-file} can also load without evaluating. If \texttt{callback} is passed to \texttt{load-data-file}, it gets each of the results. Otherwise the results are discarded (except being printed when passing \texttt{:print t}). Hence to be useful, either \texttt{load-data-file} must be called with \texttt{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 \texttt{make-load-form}, or is a quoted list, \texttt{eval} does nothing. Dumping such forms and then using using the \texttt{callback} in \texttt{load-data-file} to do some work with them is the natural way of using \texttt{dump-forms-to-file} or \texttt{with-output-to-fasl-file} and \texttt{load-data-file} to transfer large amounts of data.

Files generated by \texttt{dump-forms-to-file} or \texttt{with-output-to-fasl-file} can be loaded (by \texttt{load-data-file}) on any LispWorks platform with the same byte order. All x86/x64 architectures have the same byte order (little-endian), so \texttt{load-data-file} on any x86/x64 architecture can be used load a data file that was generated on any x86/x64 architecture. The ARM architecture has the same byte order as x86/x64. The reverse byte order (big-endian) is used by AIX and SPARC (old Solaris).
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

```
(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:

```
(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:

```
(load-data-file "my-forms.data"
  :callback 'print
  :eval nil)
```

prints this:

```
#(1 2 3)
89
(* 7 7)
QUOTE (* 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**
`with-ring-locked (ring &optional whostate timeout) &body body => result`

**Arguments**
- `ring` A ring object created by `make-ring`.
- `whostate` The status of the process while the ring is locked.
- `timeout` A timeout period, in seconds.
- `body` Lisp code.

**Values**
- `result` The result of executing `body`.

**Description**
The macro `with-ring-locked` locks the ring `ring` that during the execution of `body` no other thread can access `ring`, whether for modification or merely reading values.
whostate and timeout are used in the same way as in with-lock.

See also make-ring

without-code-coverage

Macro

Summary Prevents generation of code coverage for a body of code.

Package hcl

Signature without-code-coverage &body body => result

Arguments body Lisp forms.

Values result The result of evaluating body.

Description The macro without-code-coverage prevents generation of code coverage for the forms of body. body is evaluated as an implicit prog, except that inside body the compiler does not generate code coverage counters, unless force was supplied non-nil to generate-code-coverage or with-code-coverage-generation.

without-code-coverage is useful for error forms that you do not want to be counted.

Notes There will be a counter for the (without-code-coverage ...) form itself. If you do not want this counter, use error-situation-forms instead.

See also error-situation-forms
generate-code-coverage
with-code-coverage-generation
The HCL Package
This chapter describes the symbols in the `LINK-LOAD` package.

**Note:** this chapter applies only to LispWorks for UNIX only (not LispWorks for Linux, FreeBSD, Mac OS X or x86/x64 Solaris).

### `break-on-unresolved-functions` Function

**Package**  
`link-load`

**Signature**  
`break-on-unresolved-functions &optional stream`

**Arguments**  
`stream`  
An output stream for message reporting. If set to `nil`, then no output will be produced. By default this is `t`.

**Description**  
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.

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
This chapter applies only to LispWorks for UNIX

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

\[(\text{foreign-symbol-address } \texttt{`chmod})\]

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

\[(\text{get-foreign-symbol } \texttt{`my-func-not-yet-loaded } t)\]

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-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-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"
  "~/clc/projects/libs.a"
  "-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-for-
eign-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

Type

Summary
The 8 bit string type.

Package
lispworks

Signature
8-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…255. This is the string type that is guaranteed to always take 8 bits per element.
16-bit-string

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

Summary
Appends the contents of a file to another file.

Package  lispworks

Signature  append-file from to

Arguments
from  A pathname designator.

Append
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 265 for more information about using ASDF with LispWorks.

See also `defsystem`

### 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>
</tbody>
</table>

`base-character` is a synonym for the Common Lisp type `base-char`.

See also `base-char-code-limit`

### 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>
| Description | The function `<base-character-p>` is a predicate for base characters.  
  
  `result` is `t` if `object` is a base character, and `nil` otherwise. |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>See also</td>
<td><code>&lt;base-character&gt;</code></td>
</tr>
</tbody>
</table>

**base-char-p**

*Function*

<table>
<thead>
<tr>
<th>Summary</th>
<th>A predicate for base characters.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td><code>lispworks</code></td>
</tr>
<tr>
<td>Signature</td>
<td><code>&lt;base-char-p object =&gt; result&gt;</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>object</code> The object to be tested.</td>
</tr>
<tr>
<td>Values</td>
<td><code>result</code> A boolean.</td>
</tr>
</tbody>
</table>
| Description | The function `<base-char-p>` is a predicate for base characters, only with standard spelling.  
  
  `result` is `t` if `object` is a base character, and `nil` otherwise. |
| See also | `<base-character-p>` |

**base-char-code-limit**

*Constant*

<table>
<thead>
<tr>
<th>Summary</th>
<th>Upper bound for character codes in base characters.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td><code>lispworks</code></td>
</tr>
<tr>
<td>Description</td>
<td>The upper exclusive bound for values of <code>&lt;char-code char&gt;</code> among base characters.</td>
</tr>
</tbody>
</table>
**base-string-p**  
**simple-base-string-p**  

*Functions*

**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**  

*Type*

**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.
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.

\textbf{Compatibility note} \texttt{bmp-char} is 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}

\textbf{bmp-char-p} \textit{Function}

\textbf{Summary} The predicate for \texttt{bmp-char} objects.

\textbf{Package} \texttt{lispworks}

\textbf{Signature} \texttt{bmp-char-p object => result}

\textbf{Arguments} \textit{object} \hspace{1cm} A Lisp object.

\textbf{Values} \textit{result} \hspace{1cm} A boolean.

\textbf{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 is 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**

*Types*

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 is 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 374
**bmp-string-p**
**simple-bmp-string-p**

*Functions*

**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  
Function

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.

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~%")
    (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~%")
    (apply #'lw:call-next-advice args)
  )
\end{verbatim}

See also \texttt{defadvice}
The LISPWORKS Package

choose-unicode-string-hash-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

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 system-name &key force simulate load args target-directory}

Arguments  
- **system-name**: A symbol representing the name of the system. The system must have been defined already using the \texttt{defsystem} macro.
- **force**: 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.)
- **simulate**: 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 \texttt{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}.
- **load**: 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}.
- **args**: 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 foo.so on Linux, and on Unix it is foo.n.o where n is a platform-specific integer. On Mac OS X the object file name is foo.dylib and on Windows the object file name is foo.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

\texttt{concatenate-system output system \&key force simulate sim source-only args target-directory t-dir script-p => result}

\texttt{system ::= system-name*}

Arguments

\texttt{output} \quad The name of the required concatenated fasl.
\texttt{system-name} \quad The name of a system defined using \texttt{def-system}.
\texttt{simulate} \quad Verbosity conditions, see Description for more detail.
\texttt{sim} \quad Same as \texttt{simulate}.
\texttt{force} \quad If \texttt{t}, then all files in the system will be concatenated.
\texttt{source-only} \quad If \texttt{t}, the source files of the system are concatenated.
\texttt{target-directory} \quad The directory to search for the object files.
\texttt{t-dir} \quad Same as \texttt{target-directory}.

Values

\texttt{result} \quad A list containing the name or names of the concatenated systems.

Description

This function produces a single, concatenated fasl, \texttt{output-file}, from a list of individual systems (named amongst the \texttt{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, \texttt{output-file} must, in such cases, be a MS-Windows-compatible filename.

If \texttt{simulate} is \texttt{nil} or is not present, \texttt{concatenate-system} will work silently. Otherwise, a plan of the actions which \texttt{concatenate-system} intends to carry out is printed. What happens next depends upon the value of \texttt{simulate}:

- If it is \texttt{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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>A pathname designator.</td>
</tr>
<tr>
<td>to</td>
<td>A pathname designator.</td>
</tr>
</tbody>
</table>

**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 :supersede (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

**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:

```
(current-pathname "bar")
=> #P"C:/temp/bar.lisp"
```

While loading the binary file foo.ofasl:

```
(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:

```
(load (current-pathname "bar" nil))
```

**See also**
defsystem
pathname-location

---

**defadvice**

**Macro**

**Summary**

Defines a new piece of advice.

**Package**
lispworks
defadvice (dspec name advice-type &key where documentation)
  lambda-list &body body => nil
  
  dspec ::= fn-name | macro-name | (method generic-fn-name [(class*)])
  advice-type ::= :before | :after | :around

Arguments

  dspec
  Specifies the functional definition to which the piece of advice belongs. There are three forms which this specification may take. The first one above specifies a function by its name; the second one specifies a macro by name; the third specifies a method by the name of its generic function and by a list of classes to specialize the arguments to the method. In the case of a method the list of classes must correspond exactly to the classes of the specialized parameters of an existing method, and the advice is then attached to this method.

  When advice is provided for a macro using defadvice, 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.

  name
  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.

  advice-type
  A keyword specifying the kind of advice wanted.
where 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 :end then the advice is instead placed at the end of its section. The other permissible value for this argument is :start, which places the advice at the start of its section in the ordering (as in the default behavior).

documentation A string providing documentation on the piece of advice.

lambda-list 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.

body The main body of the advice.

Values 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.

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
### *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...
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
open
set-default-character-element-type
with-output-to-string

define-action  

Macro

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 exe-
cute-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 key-words, 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.

:oncex 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))

See also undefine-action

---

**define-action-list**

*Macro*

**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.
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 259.

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 sub-systems) 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 86. 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 eql. 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.

- **: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
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 262.

**Examples**

```lisp
(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 :lisp-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")))
)

(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 runtime 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
**Function:** deliver

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 IDE User Guide*.

**See also**

- delivered-image-p
- save-image

“Guidance for control of the storage management system” on page 112

---

**Variable:** *describe-length*

**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 \texttt{describe}

\textbf{*describe-level*} \textit{Variable}

\textbf{Summary} Controls the depth to which \texttt{describe} describes arrays, structures and conses.

\textbf{Package} \texttt{lispworks}

\textbf{Initial value} 1

\textbf{Description} The variable \texttt{*describe-level*} controls the depth to which the function \texttt{describe} describes arrays, structures and conses.

\textbf{Notes} The \texttt{describe} functionality is load-on-demand in the LispWorks image as shipped. Therefore if you have not do \texttt{(require "describe")} or called \texttt{describe}, \texttt{*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

**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 :dill-exports or by `deliver` with :dill-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**
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 32-bit LispWorks on Microsoft Windows, Intel Macintosh, Linux, x86/x64 Solaris and FreeBSD, and in 64-bit LispWorks on Windows, Intel Macintosh, Linux and x86/x64 Solaris.

See also deliver
save-image

---

dotted-list-length

Function

Summary   Similar to list-length

Package   lispworks

Signature  dotted-list-length list => result

Arguments  list          A list.

Value      result        An integer.

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**
list A list, which must be a **cons**.

**Values**
result A generalized boolean.

**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
**Variable**

*enter-debugger-directly*

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.

**Function**

evironment-variable

Summary: Reads the value of an environment variable from the environment table of the calling process.

Package: lispworks

Signature: environment-variable name => result

Arguments: name A string.

Values: result A string or nil.
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:

```
(setf (environment-variable name) 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:

```
(environment-variable "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 395
**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 UNIX/Linux/Mac OS X.

**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.
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/7-0-0-0/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`.

**Values**

- `output-truename` A pathname or `nil`.
- `warnings-p` A generalized boolean.
- `failure-p` A generalized boolean.

**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/7-0-0-0/examples/capi/applications/othello.lisp` from the LispWorks library:

```
(example-edit-file "capi/applications/othello")
```

See also
- `example-file`
- `example-compile-file`
example-load-binary-file  

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  execute-actions name-or-list &rest keyword-value-pairs &rest other-args =>

Arguments  name-or-list  An action list

keyword-value-pairs  See description.

other-args  A list.
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:

```
(action-list other-args &rest 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`:

```
(apply data 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:
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

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
ture

file-directory-p

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 "-=")
\text{T} 

CL-USER 71 > (file-directory-p ".login")
\text{NIL} 

\textbf{find-regexp-in-string} \hspace{1cm} \textit{Function}

\textbf{Summary} \hspace{1cm} Matches a regular expression.

\textbf{Package} \hspace{1cm} lispworks

\textbf{Signature}

\begin{verbatim}
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
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
  \item \textit{pattern} \hspace{0.5cm} A string or a precompiled regular expression object.
  \item \textit{string} \hspace{0.5cm} A string.
  \item \textit{start, end} \hspace{0.5cm} Bounding index designators of \textit{string}.
  \item \textit{from-end} \hspace{0.5cm} A generalized boolean.
  \item \textit{case-sensitive} \hspace{0.5cm} A generalized boolean.
  \item \textit{brackets-limits} \hspace{0.5cm} A generalized boolean.
\end{itemize}

\textbf{Values}

\begin{itemize}
  \item \textit{pos} \hspace{0.5cm} A non-negative integer or \text{nil}.
  \item \textit{len} \hspace{0.5cm} A non-negative integer or \text{nil}.
  \item \textit{brackets-limits-vector} \hspace{0.5cm} A vector.
\end{itemize}

\textbf{Description}

The function \texttt{find-regexp-in-string} searches the string \textit{string} for a match for the regular expression \textit{pattern}. The index in \textit{string} of the start of the first match is returned in \textit{pos}, and the length of the match is \textit{len}.
If from-end is nil (the default value) then the search starts at index start and ends at index end. start defaults to 0 and end defaults to nil. If from-end is true, then the search direction is reversed.

pattern should be a precompiled regular expression object or a string. If pattern is a string then find-regexp-in-string first makes a precompiled regular expression object. This operation allocates, therefore if you need to repeatedly call find-regexp-in-string with the same pattern, it is better to call precompile-regexp once and pass its result, a precompiled regular expression object, as pattern.

case-sensitive controls whether a string pattern is precompiled as a case sensitive or case insensitive search. A true value other than :default means a case sensitive search. The value nil means a case insensitive search. The default value of case-sensitive is :default which means that a string pattern is compiled with case sensitivity according to the value of the Editor variable DEFAULT-SEARCH-KIND.

When brackets-limits is non-nil, a successful call to find-regexp-in-string returns a third value which is a vector specifying the limits of matches of any pair of \( and \) 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 \( and \) is assigned a number in the order of the appearance of the \( 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

function-lambda-list

Function

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**

\textit{function} is the function whose arguments are required.

If \texttt{error-p} is \texttt{nil}, then \texttt{function-lambda-list} returns \texttt{:none} if \texttt{function} is not defined, and does not start the debugger. The default value of \texttt{error-p} is \texttt{t}, meaning that an error is signaled if \texttt{function} is undefined.

**Example**

\begin{verbatim}
TEST 2 > (function-lambda-list 'editor:create-buffer-command)
(EDITOR::P &OPTIONAL EDITOR:BUFFER-NAME)
\end{verbatim}

---

**get-inspector-values**

**sort-inspector-p**

\textit{Generic Functions}

**Summary**

Customizes the information displayed and sort order of attributes/values in the LispWorks IDE Inspector tool.

**Package**

\textit{lispworks}

**Signature**

\texttt{get-inspector-values \textit{object mode} => \textit{names, values, getter, setter, type}}

\texttt{sort-inspector-p \textit{object mode} => \textit{result}}

**Arguments**

\begin{itemize}
  \item \textit{object} \quad The object to be inspected.
  \item \textit{mode} \quad Name of a mode, or \texttt{nil}. \texttt{nil} defines the default inspection format for \textit{object}.
\end{itemize}

**Values**

\begin{itemize}
  \item \textit{names, values} \quad The two lists displayed in columns in the Inspector window.
  \item \textit{getter} \quad Ignored.
  \item \textit{setter} \quad A function used to update slot values.
  \item \textit{type} \quad Displayed in the Inspector window.
\end{itemize}
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.

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 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 UNIX/Linux/Mac OS X/FreeBSD, x86/x64 Solaris and AIX/PowerPC.

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 Unix/Linux/Mac OS X/FreeBSD 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 Unix/Linux/Mac OS X.  
"-a -a -a NUL" on Windows.

**Description**  
On Unix/Linux/Mac OS X 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*

*grep-fixed-args*  

**Variable**

Package  lispworks

Summary  Arguments added to the command string of a Grep search in the Search Files tool.

Initial value  

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*

*handle-existing-action-in-action-list*  

**Variable**

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 redefine-action.

See also:
- define-action
- execute-actions
- print-actions
- undefine-action
- redefine-action-list

*handle-missing-action-in-action-list*

Variable

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*

Variable

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 70

---

**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 nil or not present then hardcopy-system works silently. Otherwise a plan of the actions which hardcopy-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 the plan to be carried out.

:each — 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, e returns from hardcopy-system without further processing, and y and n work as expected.

Values

hardcopy-system returns nil.

Examples

(hardcopy-system 'blackboard)

(hardcopy-system 'tms :simulate :ask :command "lpr")

Notes

By default, hardcopy-system uses *print-command* as the command sent to the shell.

See also

defsystem

*print-command*

*init-file-name*

Summary

The default user initialization file.

Package

lispworks

Initial value

"~/.lispworks"
**Description**

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.

**Variable**

### *inspect-through-gui*  
**Summary**

Controls what `inspect` does in the development environment.

**Package**  
lispworks

**Initial value**

nil

**Description**

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.

**Function**

### lisp-image-name

**Summary**

Returns the name of the running image.

**Package**  
lispworks

**Signature**

`lisp-image-name => name`
Arguments None.

Values

name A string.

Description The function \texttt{lisp-image-name} returns a string representing the full path to the running \texttt{LispWorks} image. The example below is in typical \texttt{LispWorks} for Windows and \texttt{LispWorks} for Linux installations. In resaved and delivered images (including dynamic libraries such as Windows DLLs), the appropriate path is returned.

Example On Windows:

\begin{verbatim}
CL-USER 1 > (lisp-image-name)
"C:\Program Files\LispWorks\lispworks-7-0-0-x86-win32.exe"
\end{verbatim}

On Linux:

\begin{verbatim}
CL-USER 1 > (lisp-image-name)
"/usr/bin/lispworks-7-0-0-x86-linux"
\end{verbatim}

See also \texttt{*line-arguments-list*}

\texttt{*lispworks-directory*} Variable

Summary The main \texttt{LispWorks} installation directory.

Package \texttt{lispworks}

Initial value Some examples of the initial value are:

\begin{verbatim}
#P"/usr/lib/lispworks/" on Unix.

#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)
\end{verbatim}
The variable *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 lib/7-0-0-0/, then it assumes this is the LispWorks installation directory and sets *lispworks-directory* accordingly. Additionally, LispWorks for Macintosh running on Cocoa looks for such a subdirectory in the Library folder alongside its application bundle, and if found it sets *lispworks-directory* accordingly.

On non-Windows platforms, LispWorks then consults the Unix environment variable LISPWORKS_DIRECTORY. If this is set, then *lispworks-directory* is set accordingly.

The lib/7-0-0-0/ subdirectory of *lispworks-directory* should include these subdirectories:

- **config**, which contains the configuration files.
- **patches**, which contains any public (numbered) patches that are distributed by LispWorks Ltd.
- **private-patches**, which is the place to put private (named) patches that are sent to you by Lisp Support.
- **postscript**, which contains configuration files for printing using the CAPI printing library. See “Configuring the printer” on page 157 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, and on Unix it is foo.o 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 make-unregistered-action-list &key documentation sort-time dummy-actions default-order execution-function =>
Arguments

- **documentation**: A string.
- **sort-time**: One of `:execute` or `:define-action`.
- **dummy-actions**: A list.
- **default-order**: A list.
- **execution-function**: A function.

Description

Return an action-list not registered in the global registry of lists. The keyword arguments are as for `define-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 user-defined function accepting arguments of the form:

```
(the-action-list other-argv-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 also define-action-list
*handle-warn-on-redefinition*

**make-mt-random-state**  
*Function*

**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*

**Summary**
Returns a pseudo-random number using the Mersenne Twister algorithm.

**Package**
lispworks

**Signature**
`mt-random arg &optional state => random-number`

**Arguments**
- `arg` A positive [integer](#) or a positive [float](#).
- `state` An object of type [mt-random-state](#). The default is the value of [*mt-random-state*](#).

**Values**
- `random-number` A non-negative number less than `arg` and of the same type as `arg`.

**Description**
The function `mt-random` returns a pseudo-random number which is non-negative, less than `arg` and is of the same type as `arg`.


We thank the authors for making the algorithm freely available.

`mt-random` is analogous to [cl:random](#).

**See also**
- make-mt-random-state
- [*mt-random-state*](#)

**Variable**

*mt-random-state* 

**Summary**
The default random state used by `mt-random`.
<table>
<thead>
<tr>
<th><strong>Package</strong></th>
<th>lispworks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>The variable <em>mt-random-state</em> contains an object of type mt-random-state which is the default state used by mt-random if a state is not supplied. <em>mt-random-state</em> is analogous to cl:<em>random-state</em>.</td>
</tr>
</tbody>
</table>

See also: make-mt-random-state  
mt-random  
mt-random-state

### mt-random-state

<table>
<thead>
<tr>
<th><strong>Type</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Package</strong></th>
<th>lispworks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>The Mersenne Twister pseudo-random number generator uses state data contained in a object of type mt-random-state. mt-random-state is analogous to cl:random-state.</td>
</tr>
</tbody>
</table>

See also: *mt-random-state*  
mt-random  
mt-random-state-p

### mt-random-state-p

<table>
<thead>
<tr>
<th><strong>Function</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Package</strong></th>
<th>lispworks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Signature</strong></td>
<td>mt-random-state-p arg =&gt; result</td>
</tr>
</tbody>
</table>

---

1018
Arguments

arg

An object.

Values

result

A boolean.

Description

The function \texttt{mt-random-state-p} returns \texttt{t} if \texttt{arg} is an object of type \texttt{mt-random-state}, and \texttt{nil} otherwise.

\texttt{mt-random-state-p} is analogous to \texttt{cl:random-state-p}.

See also

\texttt{mt-random-state}

\textbf{pathname-location} \hfill \textit{Function}

Summary

Returns the location of a file.

Signature

\texttt{pathname-location pathname \Rightarrow location}

Arguments

pathname

A pathname designator.

Values

location

A pathname.

Description

The function \texttt{pathname-location} returns a pathname \texttt{location} that represents the directory where the file \texttt{pathname} resides. Each of the name, type and version components of \texttt{location} are \texttt{nil}.

Example

Due to the ANSI Common Lisp definition of the \texttt{directory} function and the fact that LispWorks returns fully specified truenames, the form

\begin{verbatim}
(directory (truename "/tmp/"))
\end{verbatim}

will always signal an error or return the list \texttt{(#P="/tmp/")}. To obtain the contents of the \texttt{/tmp} directory, use the form

\begin{verbatim}
(directory (pathname-location (truename "/tmp/")))
\end{verbatim}
precompile-regexp

Function

Summary
Precompiles a regular expression object.

Package
lispworks

Signature
precompile-regexp string => pattern

Arguments
string A string.

Values
pattern A precompiled regular expression object.

Description
The function `precompile-regexp` returns a precompiled regular expression object suitable for passing as `pattern` to functions like `find-regexp-in-string`.

Notes
For the regular expression syntax, see `find-regexp-in-string`.

See also
`find-regexp-in-string`
`regexp-find-symbols`
`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
dubya CL-USER 3 >

**quit**

Function

Summary Quits LispWorks.

Package lispworks

Signature quit &key status confirm ignore-errors-p return
Arguments

- \textit{status} \hspace{1cm} An integer.
- \textit{confirm} \hspace{1cm} A generalized boolean.
- \textit{ignore-errors-p} \hspace{1cm} A generalized boolean.
- \textit{return} \hspace{1cm} A generalized boolean.

Values

- \textit{quit} does not return, or returns \textit{t}.

Description

The function \textit{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 "\textit{Confirm when quitting image}" are run. If any action item returns \textit{nil}, then LispWorks does not exit.

2. Otherwise, if \textit{confirm} is true (the default value is \textit{nil}) then a question like
   "\textit{Do you really want to exit LispWorks?}"
   is presented to the user. If the answer \textit{No} is supplied, then LispWorks does not exit. Otherwise, the action items of the action list "\textit{When quitting image}" are run, and then LispWorks exits, and the value \textit{status} is returned to the Operating System as the exit value of the LispWorks process. The default value of \textit{status} is 0.

If \textit{ignore-errors-p} is true, then any error signaled during the running of the action list items or the confirm prompt is ignored and \textit{quit} proceeds to exit the image. If \textit{ignore-errors-p} is \textit{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 \textit{ignore-errors-p} is \textit{nil}.

If \textit{return} is true and LispWorks is going to exit, then \textit{quit} returns \textit{t}. This can be used if you want some other Lisp process 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 \textit{return} is \texttt{nil} then \texttt{quit} does not return. The default value of \textit{return} is \texttt{nil}.

Notes On Cocoa, when you define your own application menu (by passing \texttt{:application-menu} when making the application interface), the \texttt{Quit} menu item needs to call \texttt{capi:destroy} on the application interface, rather than \texttt{quit}. See \texttt{capi:cocoa-default-application-interface} in the \texttt{CAPI User Guide and Reference Manual} for more information.

See also \texttt{save-image}

\textbf{rebinding} \quad \textit{Macro}

Summary Ensures unique names for all the variables in a groups of forms.

Package \texttt{lispworks}

Signature \texttt{rebinding (&rest vars) &body body => form}

Arguments
vars \quad \text{The variables to be rebound.}
body \quad \text{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 \textit{body} wrapped in a form which creates a unique name for each of the variables (compare with \texttt{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
(defmacro 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
remove-advice dspec name => nil
dspec ::= fn-name |
macro-name |
(method generic-fn-name [(class*)]])
Arguments  

\textit{dspec}  

Specifies the functional definition to which the piece of advice belongs. The specification contains the name of the associated function. In the case of a method the list of classes is used to identify from which particular method the advice should come. This list must correspond exactly with the classes corresponding to the specialized parameters for some method belonging to the generic function.

\textit{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  

\textbf{remove-advice} returns \texttt{nil}.

Description  

\textbf{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 \textbf{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 \textbf{remove-advice}, except that you do not need to quote the arguments.

Notes  

\textbf{remove-advice} is an extension to Common Lisp.

See also  

\texttt{defadvice}  
\texttt{delete-advice}
**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.
### round-to-single-precision Function

**Summary**
Rounds the given float to single-precision format (32 bits) and returns it as a `double-float` (64 bits).

**Package**
lispworks

**Signature**
`round-to-single-precision float => double-float`

**Arguments**
- float
  
**Values**
- `double-float`

**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.

The default size on Windows and Linux is 64 bits as specified by the IEEE standard.

LispWorks supports 3 floating point formats, `short-float`, `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 doing

```
(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, three 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**
- value: The character in string at index.

**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

**Summary**

Configures the value of `lw:*default-character-element-type*`.

**Package**

lispworks

**Signature**

`set-default-character-element-type type => type-defaults`

**Arguments**

- **type**
  
  A character type. This can take any of the values `cl:base-char`, `bmp-char` and `cl:character`. For backwards compatibility, `simple-char` is also allowed, and is treated as if `cl:character` was passed.

**Values**

- **type-defaults**
  
  The new value of `lw:*default-character-element-type*`.

**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.
set-quit-when-no-windows

Function

Summary
Overrides the \texttt{:quit-when-no-windows} keyword argument to \texttt{deliver}. 

Signature
\texttt{set-quit-when-no-windows on}

Arguments
\texttt{on} \hspace{1cm} \texttt{nil, t or the keyword :check}

Description
The function \texttt{set-quit-when-no-windows} can be used at runtime in a delivered application to override the value of the \texttt{:quit-when-no-windows} keyword to \texttt{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 \texttt{on} is \texttt{nil}, then the application will not quit merely because there are no remaining open windows.

If \texttt{on} is \texttt{t}, then the application will quit when there are no remaining open windows after the application has opened at least one CAPI window.

If \texttt{on} is \texttt{:check}, then the application will quit immediately if there are no open windows at the current time. Unlike with \texttt{: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 \texttt{on} is \texttt{t}. 

See also
\begin{verbatim}
string
open
*default-character-element-type*
with-output-to-string

“Unicode support” on page 373
“Controlling string construction” on page 380
\end{verbatim}
See also :quit-when-no-windows in the LispWorks Delivery User Guide

simple-char

Summary The simple character type. simple-char is deprecated.

Package lispworks

Signature simple-char

Description 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

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**

`split-sequence-if predicate sequence &key start end key coalesce-separators => result`

**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` returns a list of subsequences of `sequence` (between `start` and `end`), split by where the function `predicate` returns true 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-not

split-sequence-if-not

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**  
\[ \text{start-tty-listener} \, \text{force} = \rightarrow \text{process} \]

**Arguments**  
force  
A generalized boolean.

**Values**  
process  
A listener process, or \texttt{nil}.

**Description**  
The function \texttt{start-tty-listener} returns a process that runs a listener read-eval-print loop connected to \texttt{*terminal-io*}.

If \texttt{force} is \texttt{nil}, then \texttt{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, \texttt{start-tty-listener} simply returns \texttt{nil} and does not start a new process. If no such process exists, or if \texttt{force} was \texttt{t}, then \texttt{start-tty-listener} starts a new listener process named "TTY Listener", and returns it.

If a REPL with I/O through \texttt{*terminal-io*} (such as a REPL started by \texttt{start-tty-listener}) is in the debugger, then by default it blocks multiprocessing. This behavior is controlled by the value of \texttt{*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**  

<table>
<thead>
<tr>
<th>string</th>
<th>A simple-text-string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>An index.</td>
</tr>
</tbody>
</table>

**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

```
(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

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
```
(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

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 374

**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**

**Package**  
{lispworks}

**Signature**  
{true &rest ignore => t}

**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**  
undefined-action-list  

**Arguments**  
uid  
A lisp object.

**Values**  
None.

**Description**  
The macro **undefined-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  

**Arguments**  
char  
A character  
style  
A keyword

**Values**  
flag  
A generalized boolean
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**

**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.
**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`
Functions

\textbf{unicode-char-equal} \quad \textbf{unicode-char-not-equal}

\textbf{Summary} \quad \text{Compares two characters, ignoring case using specified Unicode rules.}

\textbf{Package} \quad \text{lispworks}

\textbf{Signatures}

\texttt{unicode-char-equal char1 char2 &key style} \Rightarrow \texttt{flag}

\texttt{unicode-char-not-equal char1 char2 &key style} \Rightarrow \texttt{flag}

\textbf{Arguments}

\begin{itemize}
  \item \texttt{char1} \quad \text{A character}
  \item \texttt{char2} \quad \text{A character}
  \item \texttt{style} \quad \text{A keyword}
\end{itemize}

\textbf{Values}

\texttt{flag} \quad \text{A generalized boolean}

\textbf{Description}

The function \texttt{unicode-char-equal} returns true if the characters \texttt{char1} and \texttt{char2} are equal, and the function \texttt{unicode-char-not-equal} returns true if the characters \texttt{char1} and \texttt{char2} are not equal. Both functions ignore case using Unicode rules specified by \texttt{style}.

The current implementation only supports one style of comparison:

\begin{itemize}
  \item \texttt{:simple-case-fold}
    \quad \text{Compared characters using the simple case folding rules in Unicode 6.3.0.}
\end{itemize}

\textbf{See also} \quad \texttt{unicode-char-greaterp}
Functions

**Summary**

Compares two characters, ignoring case using specified Unicode rules.

**Package**

lispworks

**Signatures**

`unicode-char-greaterp char1 char2 &key style => flag`

`unicode-char-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-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`

---

1051
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`
unicode-string-greaterp
unicode-string-lessp

Functions

Summary
Compares two strings, ignoring case using specified Unicode rules.

Package
lispworks

Signatures
unicode-string-greaterp string1 string2 &key start1 start2
df end1 end2 style => mismatch-index

unicode-string-lessp string1 string2 &key start1 start2 end1
df 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
c1: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 com-
parison:
: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`
- `unicode-string-not-lessp`

### Functions

**`unicode-string-not-greaterp`**

**`unicode-string-not-lessp`**

#### Summary

Compares two strings, ignoring case using specified Unicode rules.

#### Package

`lispworks`

#### Signatures

- `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`. 
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`

### user-preference

**Function**

**Summary**

Gets or sets a persistent value in the user’s registry.

**Package**

`lispworks`

**Signature**

```
user-preference path value-name &key product => value, valuep
```

**Signature**

```
(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 Unix/Linux/Mac OS X 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.

3. The CAPI provides a way to store window geometry — see the reference entry for *capi:top-level-interface-save-geometry-p* in the *CAPI User Guide and Reference Manual*.

Example

This example is on Microsoft Windows. Note the use of backslashes as directory separators in the *path* argument:

```lisp
(setq (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:

```lisp
(setq (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)
=>
t

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.
This macro executes the body of code if the form evaluates to a non-nil result. Within the body, the variable \textit{var} is bound to the result of the \textit{form}.

The form

\begin{verbatim}
(when-let (position (search string1 string2))
  (print position))
\end{verbatim}

macroexpands to

\begin{verbatim}
(let ((position (search string1 string2)))
  (when position
    (print position)))
\end{verbatim}

See also \textit{when-let*}

\begin{center}
\textbf{when-let*}
\end{center}

\textit{Macro}

\textbf{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.

\textbf{Package} lispworks

\textbf{Signature} \texttt{when-let* bindings \&body body => result}

\texttt{bindings ::= (\{var form\} \*)}

\textbf{Arguments} \begin{itemize}
  \item \textit{var} A variable whose value is used in the evaluation of \textit{body}.
  \item \textit{form} A form, which must evaluate to non-nil.
  \item \textit{body} A body of code to be evaluated conditionally on the result of \textit{form}.
\end{itemize}

\textbf{Values} \begin{itemize}
  \item \textit{result} The result of evaluating \textit{body} using the value \textit{var}.
\end{itemize}
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

```lisp
(defun 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 [whitespace1], 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  
(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**

```lisp
(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 default-constructor-arguments 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 default-constructor-arguments. See the entry for default-constructor-arguments 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 default-constructor-arguments is used, but not much. If you find it convenient, there is no reason not to use it.

See also default-constructor-arguments
call-java-method-error
“Defining specific callers” on page 170

call-java-method-error

Summary call-java-method failed to find the method.

Package lw-ji

Superclasses java-interface-error

Subclasses None.
The condition class `call-java-method-error` is signaled when `call-java-method` failed to find the method.

See also `call-java-method`

catching-java-exceptions

catching-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 `catching-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`, `catching-java-exceptions` returns two values: `nil` and the Java exception object (analogous to `cl:ignore-errors`).

The macro `catching-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. `catching-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 find-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 t 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 173

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
Arguments

- **jobject**
  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* `jobject`.

`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 185

### Functions

#### create-instance-jobject-list

Construct a *jobject* for a CLOS instance.

**Package**

lw-ji

**Signature**

- `create-instance-jobject-list instance args => jobject`
- `create-instance-jobject 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-jobject-list** and **create-instance-jobject** ignore the current **jobject** in **instance**, if there is one. There is no problem calling **create-instance-jobject-list** and **create-instance-jobject** 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 \texttt{create-java-object-error}, which reports the actual failure.

Notes \texttt{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 \texttt{define-java-constructor} is used, but not much. If you find it convenient, there is no reason not to use it.

See also \texttt{define-java-constructor}

\texttt{create-java-object-error} \hspace{1cm} \textit{Condition}

Summary \texttt{create-java-object} failed to find constructors.

Package \texttt{lw-ji}

Superclasses \texttt{java-interface-error}

Subclasses None.

Description The condition class \texttt{create-java-object-error} is signaled when \texttt{create-java-object} failed to find constructors.

See also \texttt{create-java-object}

\texttt{default-constructor-arguments} \hspace{1cm} \textit{Generic Function}

Summary Returns a default list of arguments to pass to the constructor.

Package \texttt{lw-ji}

Signature \texttt{default-constructor-arguments instance}
Method signatures

\[
\text{default-constructor-arguments } (\text{instance \ standard-java-object}) => \text{nil}
\]

Arguments

\[\text{instance} \quad \text{An instance of a subclass of \text{standard-java-object}.}\]

Description

The generic function \text{default-constructor-arguments} returns a default list of arguments to pass to the constructor. It is used by \text{create-instance-jobject-list} when its \text{args} argument is \text{t}. \text{default-constructor-arguments} is also used by the \text{cl:initialize-instance} method of \text{standard-java-object} when \text{:construct} is passed with value \text{t}. It is intended for you to specialize on your own classes.

The default method returns \text{nil}, which is sometimes useful, but in most cases you probably need to pass something.

See also \text{create-instance-jobject-list}

\textbf{default-name-constructor}

Function

Summary

The default \text{name-constructor} used by the importing interface.

Package

\text{lw-ji}

Signature

\text{default-name-constructor} \ prefix \ method-or-field-name => \ symbol-name

Arguments

\[\text{prefix} \quad \text{A string.}\]

\[\text{method-or-field-name} \quad \text{A string.}\]

Values

\[\text{symbol-name} \quad \text{A string.}\]

Description

The function \text{default-name-constructor} is the default \text{name-constructor} used by the importing interface. See \text{gener-}
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
Failed to find the field, or found the field but wrong static-\textit{p} value.

\texttt{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 \texttt{define-field-accessor} forms for public fields.

See also \texttt{setup-field-accessor}

\textbf{define-java-caller}

\textbf{define-java-constructor}

\textit{Macros}

Summary Define a Java caller, which is a function that calls a Java method or a constructor.

Package \texttt{lw-ji}

Signature \texttt{define-java-caller} \texttt{name class-name method-name \&key signatures \=> result}

Signature \texttt{define-java-constructor} \texttt{name class-name \&key class-symbol signatures \=> result}

Arguments \begin{itemize}
  \item \texttt{name} \hspace{1cm} A symbol.
  \item \texttt{class-name} \hspace{1cm} A string.
  \item \texttt{method-name} \hspace{1cm} A string.
  \item \texttt{signatures} \hspace{1cm} A list of strings.
  \item \texttt{class-symbol} \hspace{1cm} A symbol.
\end{itemize}

Values \begin{itemize}
  \item \texttt{result} \hspace{1cm} \texttt{name} or \texttt{nil}.
\end{itemize}
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` 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 166), 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{jobject} 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{jobject} representing the Java object, or \texttt{nil} if the method returned \texttt{null}. Constructors always return \texttt{jobject}s.

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-class-definitions-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 runtime 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-class-definitions-to-file`
- `import-java-class-definitions`
- `verify-java-callers`
- `verify-java-caller`
- “Defining specific callers” on page 170

**define-java-callers**

Macro

Summary

Define multiple Java callers for methods in the same class.
The LW-JI Package

Package lw-ji

Signature define-java-callers class-name &body method-specs => class-name

Arguments

- class-name A string.
- method-specs Lists.

Values class-name A string.

Description 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 default-constructor-arguments.

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 default-constructor-arguments:

\[
\text{(define-java-callers class-name (caller-name1 method-name1) (caller-name2 method-name2)) =>}
\]
\[
\text{(progn (define-java-caller caller-name1 class-name method-name1) (define-java-caller caller-name2 class-name method-name2) class-name)}
\]

define-java-callers is a more compact way to write several methods for the same class, but functionally it is identical to using default-constructor-arguments explicitly.

define-java-callers returns the class-name.

See also default-constructor-arguments
**define-lisp-proxy**  

**Macro**

**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.

**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{jobject}s 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\texttt{-}object\texttt{-}ensure\texttt{-}global}, and then it can be used out of scope.

The default value of \texttt{:jobject-scope} is \texttt{:global}.

\subsection*{user-data}

The \texttt{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 \texttt{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 \texttt{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 runtime 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 166.

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 user-data. If :with-user-data was used in the method specification then use its value, otherwise if :with-user-data was used in :options item use this value, otherwise default to nil, then
   d) Apply the function: if using user-data, apply the function to the 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 default-function itself as the function to call, then
   b) Check whether need to pass user-data, which is specified by the :default-function-with-user-data in the :options item, then
   c) Apply the function: if user-data needs to be used, apply the function to the user-data, method-name and the Lisp arguments. Otherwise apply the function to the 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 runtime, 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-object-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-object-scope :local or nil will reduce the overhead significantly.

Examples

(exexample-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 superclasses as needed based on the tree.

**Package**  
lw-ji

**Signature**  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lisp-name</td>
<td>A symbol.</td>
</tr>
<tr>
<td>java-class-tree</td>
<td>A tree.</td>
</tr>
<tr>
<td>force-p</td>
<td>A generalized boolean.</td>
</tr>
</tbody>
</table>

**Values**  

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>A class metaobject.</td>
</tr>
</tbody>
</table>

**Description**  
The function `ensure-lisp-classes-from-tree` creates a class for `lisp-name`, and potentially some or all the superclasses 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 runtime.

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 sub-class of standard-java-object.

Package lw-ji

Signature ensure-supers-contain-java.lang.object super-symbols lisp-name => nil
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 166), 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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java-class-name</td>
<td>A string.</td>
</tr>
<tr>
<td>lisp-name</td>
<td>A symbol.</td>
</tr>
<tr>
<td>package-name</td>
<td>A string.</td>
</tr>
<tr>
<td>prefix</td>
<td>A string or nil</td>
</tr>
<tr>
<td>name-constructor</td>
<td>A function designator.</td>
</tr>
<tr>
<td>export-p</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>create-defpackage</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>lisp-class-p</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>lisp-supers</td>
<td>A list of symbols.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list-of-definitions</td>
<td>A list.</td>
</tr>
<tr>
<td>lisp-name-symbol</td>
<td>A symbol.</td>
</tr>
<tr>
<td>package-name-string</td>
<td>A package name.</td>
</tr>
</tbody>
</table>

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 java-
class-name. lisp-name can also be nil.

If lisp-name is not supplied, the system creates a Lisp symbol
based on the java-class-name. Note that this is different from
passing nil, because in the latter case lisp-name stays nil.

package-name is used only if lisp-name is supplied as 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 lisp-name is nil and package-name is nil or not sup-
plied, the current package is used.

prefix, if supplied, specifies a prefix to use for the names of
the definitions. If prefix is not supplied or is nil, the name of
the Java class without the package part is uppercased and
used as the prefix (for example for "java.io.File" the prefix is
"FILE"). The prefix is passed to the name-constructor to con-
struct names for the Java callers.

If name-constructor is supplied, it must be a function taking
two string arguments: the 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 (with-
out changing the case) in the package to create the symbol
that is used as the name of the caller. The name-constructor
defaults to a function (default-name-constructor) that
concatenates the 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 prefix would generate "FILE.EXISTS".

export-p controls whether all the Java callers are exported
from the package. If it is t all the Java callers are exported,
otherwise they are not. The default of export-p is 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 1100.

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
default-constructor-arguments, 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

\[\text{(define-java-constructor FILE.NEW "java/io/File")}\]

The caller for the method "exists" would be defined by:

\[\text{(define-java-caller FILE.EXISTS "java/io/File" "exists")}\]

the accessor for for the field "separator" would be defined by:

\[\text{(define-field-accessor file.separator "java/io/File" "separator" t t)}\]

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:setf definition to allow using cl:setf 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 185.

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 169

---

**get-java-virtual-machine**

*Function*

Summary If a Java virtual machine has started, return it.

Package lw-ji

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. This uses **JNI_GetCreatedJavaVMs**.
Compatibility note
At the time of writing, `get-java-virtual-machine` is not available in LispWorks 7.0.

See also `init-java-interface`

<table>
<thead>
<tr>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>get-jobject</strong></td>
</tr>
<tr>
<td><strong>ensure-is-jobject</strong></td>
</tr>
</tbody>
</table>

**Summary**
Get the `jobject` of the argument.

**Package**
lw-ji

**Signature**
`get-jobject object => object`

**Signature**
`ensure-is-jobject object caller => object`

**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, `get-jobject` returns `nil` but `ensure-is-jobject` signals an error. `ensure-is-jobject` uses `caller` in the error message to identify where the error occurred.

**Notes**
`get-jobject` is the predicate to check whether an object is a Java object.

See also `jobject-p` `jobject`
**get-primitive-array-region**  
**set-primitive-array-region**  

*Functions*

**Summary**  
Copy between a Java array of primitive type and a buffer specified by a foreign pointer.

**Package**  
**lw-ji**

**Signature**  
**get-primitive-array-region**  
array &key start end buffer buffer-size => target-buffer, foreign-type

**Signature**  
**set-primitive-array-region**  
array buffer &key start end => t, foreign-type

**Arguments**  
array A Java array of primitive type.  
start, end Bounding index designators for array.  
buffer An FLI pointer.  
buffer-size A non-negative integer.

**Values**  
target-buffer buffer or a new buffer.  
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 lisp-array-to-primitive-array or primitive-array-to-lisp-array instead.

See also

lisp-array-to-primitive-array
primitive-array-to-lisp-array
“Working with Java arrays” on page 178
**get-superclass-and-interfaces-tree**

*Function*

**Summary**

Returns the superclasses and implemented interfaces of a supplied Java class.

**Package**

lw-ji

**Signature**

`get-superclass-and-interfaces-tree java-class => java-class-tree`

**Arguments**

`java-class` A *jobject*.

**Values**

`java-class-tree` A tree.

**Description**

The function `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 `ensure-lisp-classes-from-tree`. It may be useful on its own, as a quick way of finding the tree for a class.

`java-class` must be a Java class, that is a *jobject* corresponding to a class. Typically that would be the result of `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", `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
lw-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
lisp-name-symbol A symbol.

Description
The macro import-java-class-definitions generates all the definitions for the class java-class-name, and wraps cl:progn around them, and returns this from the macroexpansion. Therefore evaluation of an import-java-class-definitions form defines all the callers for java-class-name.

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 `generate-java-class-definitions`, and all the keywords are passed to it. See the entry for `generate-java-class-definitions` for the effects of the keywords.

During macroexpansion, `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 `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 `lisp-name-symbol` is the Lisp name (which is `lisp-name`, or is generated by `generate-java-class-definitions` if `lisp-name` was not supplied).

See “Importing classes” on page 169 for discussion.

Notes

1. You can avoid the need for running Java during compilation by writing the definitions using the writers (`write-java-class-definitions-to-file` or `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

```
(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 169
init-java-interface

Function

Summary Initializes the Java interface.

Package lw-ji

Signature

\[ \text{init-java-interface} \& \text{key jvm-library-path java-class-path option-strings jni-env-finder java-virtual-machine class-finder class-loader-finder java-to-lisp-debugger-hook report-error-to-java-host send-message-to-java-host} \Rightarrow \text{result} \]

Arguments

- \text{jvm-library-path}: A string, \text{t} or \text{nil}.
- \text{java-class-path}: A string or a list of strings.
- \text{option-strings}: A list.
- \text{jni-env-finder}: A function designator, or \text{nil}.
- \text{java-virtual-machine}: A foreign pointer of type \text{java-vm-poi}, or \text{t}.
- \text{class-finder}: A function designator or \text{nil}.
- \text{class-loader-finder}: A function designator or \text{nil}.
- \text{java-to-lisp-debugger-hook}: A function designator or \text{nil}.
- \text{report-error-to-java-host}: A function designator or \text{nil}.
- \text{send-message-to-java-host}: A function designator or \text{nil}.

Values

- \text{result}: \text{t} or the keyword \text{:no-java-to-lisp}.

Description

The function \text{init-java-interface} needs to be called before using any of the runtime part of the Java interface. That includes the interface functions that are documented as requiring Java, and any of the user-defined callers. The defin-
ers in general do not need running Java, but the importing interface does.

**Note:** On Android, **init-java-interface** is called automatically by the system initialization, so you do not need to (and must not) call it.

**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 **init-java-interface** needs to initialize the JVM, it must be called with **jvm-library-path** either `t` or the path of a dynamic library, and **jni-env-finder** must be `nil`. When **jvm-library-path** is `t`, **init-java-interface** uses a default library path, which is currently "jvm.dll" on Microsoft Windows, 
"/System/Library/Frameworks/JavaVM.framework/JavaVM" on Mac OS X, and "-ljvm" on other Unix variants. The library must implement the JVM, which means exporting the JNI functions. 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** `t` 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 "/myhome/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.lisp-`
works.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 C 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 jvm-library-path must be nil, and either jni-env-finder or java-virtual-machine must be supplied as non-nil.

If jni-env-finder is used it must be a function of no arguments and 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 needs to cope with being called on any thread, and the result needs to be valid until the thread dies, and implementing code must deal with eliminating it when the thread goes. 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. It can be either t, which tells the system to find the Java Virtual Machine, or a foreign pointer of type java-vm-poi, which
corresponds to the C type \texttt{JavaVM*}. If \texttt{java-virtual-machine} is \texttt{t}, the system finds the virtual machine using \texttt{JNI\_GetCreatedJavaVMS}, otherwise it uses the pointer as the virtual machine.

\textbf{Notes}

The simple option when the JVM is already running is just passing \texttt{java-virtual-machine t}. However, the function that the system uses, \texttt{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 \texttt{init-java-interface} with \texttt{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 \texttt{init-java-interface} either because you got it from code that started the Java VM (by \texttt{JNI\_CreateJavaVM}), or by exporting \texttt{JNI\_OnLoad} from a dynamic library. However, it is not a good idea to export \texttt{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 184.

\textbf{Compatibility note}

At the time of writing, the keyword argument \texttt{java-virtual-machine} is not available in LispWorks 7.0

\textbf{Description (cont.)}

\texttt{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 \texttt{jobject} representing a class for this string (for example, a caller for the method \texttt{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, \texttt{class-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 **object**. 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 **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 `cl:*terminal-io*`, which may be useful enough when just debugging.

`send-message-to-java-host`, when supplied, must be a function of two arguments: a string which is the message and a keyword, one of `:append`, `:prepend`, `:append-no-scroll` and `:reset`, which tells it how to deal with it. If the keyword is `:reset`, the "messages output" should be reset, otherwise the string should be added to the "messages output" as appropriate. 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 `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.

`init-java-interface` returns either `t` for success, or `: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 `com.lispworks.LispCalls`.

**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  \texttt{intern-and-export-list} returns \texttt{nil}.

Description  The function \texttt{intern-and-export-list} finds the package specified by \texttt{package-name}, interns all the strings in this package, and exports the resulting symbols.

\texttt{intern-and-export-list} is a utility function that is used by the importing interface to export symbols by default (when not using \texttt{defpackage}).

See also  \texttt{generate-java-class-definitions}

\texttt{jaref} \hspace{1cm} \textit{Function}

Summary  Read and set an element in a Java array.

Package  lw-ji

Signature  \texttt{jaref array \&rest indices => element}

\hspace{1cm} (\texttt{setf jaref}) \hspace{0.5cm} \texttt{new-value array \&rest indices => new-value}

Arguments

\begin{itemize}
  \item \texttt{array}  \hspace{0.5cm} A Java array, of any type.
  \item \texttt{indices}  \hspace{0.5cm} Non-negative integers.
  \item \texttt{new-value}  \hspace{0.5cm} A valid element for \texttt{array}.
\end{itemize}

Values  \texttt{element}  \hspace{0.5cm} A Lisp object, a \texttt{jobject} or \texttt{nil}.

Description  The function \texttt{jaref} reads and sets the value of an element in the Java array \texttt{array}.

Each of the \texttt{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 \texttt{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 178

java-array-element-type

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 166.
- 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.

See also

- `java-primitive-array-element-type`
- `java-object-array-element-type`
- “Working with Java arrays” on page 178

Condition

`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  Condition

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**

*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 178

**java-array-simple-error**

*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-caller` and `java-array-error-array` on these conditions.

### See also
`java-array-error`

### `java-bad-jobobject` *Condition*

#### Summary
An abstract class, meaning that it is never 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-jobject-error`
- `java-not-an-array-error`

#### Readers
- `java-bad-jobobject-caller`
- `java-bad-jobobject-object`

#### Description
The condition class `java-bad-jobobject` 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 runtime 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

Condition

Summary The superclass of all conditions that are signaled for Java exceptions.

Package lw-ji

Superclasses None

Subclasses java-normal-exception
java-serious-exception

Readers java-exception-string
java-exception-exception-name
java-exception-java-backtrace
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-jobobject-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`
### The LW-JI Package

#### java-instance-without-jobject-error

<table>
<thead>
<tr>
<th>Superclasses</th>
<th>java-bad-object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subclasses</td>
<td>None.</td>
</tr>
<tr>
<td>Description</td>
<td>The condition class <code>java-instance-without-jobject-error</code> is signaled when an argument that needs to be a Java object is an instance of <code>standard-java-object</code> that does not have a <code>jobject</code>.</td>
</tr>
<tr>
<td>Notes</td>
<td>You can use the <code>java-bad-jobject</code> readers <code>java-bad-job-ject-caller</code> and <code>java-bad-jobject-object</code> on these conditions.</td>
</tr>
</tbody>
</table>
| See also     | `java-bad-jobject`  
               `java-not-a-java-object-error`  
               `java-not-an-array-error` |

#### java-interface-error

| 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-jobject`  
               `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.

Package lw-ji

Superclasses java-exception

Subclasses field-exception java-method-exception

Description Conditions of class java-normal-exception are signaled for exceptions that are "normal", that is they do not suggest that the system is broken.
**java-not-a-java-object-error**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Conditions signaled when an argument that needs to be a Java object is not a <code>jobject</code> or an instance of <code>standard-java-object</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>lw-ji</td>
</tr>
<tr>
<td>Superclasses</td>
<td>java-bad-jobobject</td>
</tr>
<tr>
<td>Subclasses</td>
<td>None.</td>
</tr>
<tr>
<td>Description</td>
<td>The condition class <code>java-not-a-java-object-error</code> is signaled when an argument that needs to be a Java object is not a <code>jobject</code> or an instance of <code>standard-java-object</code>.</td>
</tr>
<tr>
<td>Notes</td>
<td>You can use the <code>java-bad-jobobject</code> readers <code>java-bad-jobobject-caller</code> and <code>java-bad-jobobject-object</code> on these conditions.</td>
</tr>
<tr>
<td>See also</td>
<td>java-bad-jobobject java-instance-without-jobobject-error java-not-an-array-error</td>
</tr>
</tbody>
</table>

**java-not-an-array-error**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Conditions signaled when an argument that needs to be an array of some type is not an array of the expected type.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>lw-ji</td>
</tr>
<tr>
<td>Superclasses</td>
<td>java-bad-jobobject</td>
</tr>
<tr>
<td>Subclasses</td>
<td>None.</td>
</tr>
</tbody>
</table>
Description
The condition class \texttt{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 \texttt{java-bad-object} readers \texttt{java-bad-object-caller} and \texttt{java-bad-object-object} on these conditions.

See also
\texttt{java-bad-object}
\texttt{java-not-a-java-object-error}
\texttt{java-instance-without-object-error}

\begin{description}
\item[Function] \texttt{java-object-array-element-type}
\item[Summary] Returns the element type of a Java array of a non-primitive element type.
\item[Package] \texttt{lw-ji}
\item[Signature] \texttt{java-object-array-element-type object => result}
\item[Arguments] \texttt{object} A Java object.
\item[Values] \texttt{result} One of the keywords \texttt{:array}, \texttt{:object} and \texttt{:string}, or \texttt{nil}.
\item[Description] The function \texttt{java-object-array-element-type} returns the element type (as a keyword as listed in “Types and conversion between Lisp and Java” on page 166) if \texttt{object} is an array with non-primitive element type. If \texttt{object} is some other type of Java object, \texttt{java-object-array-element-type} returns \texttt{nil}. Otherwise it signals an error.
\end{description}
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 178

### 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 166, “CLOS partial integration” on page 185
**Conditions**

**java-out-of-bounds-error**

**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

\textit{result} \quad A \ keyword, \texttt{t} \ or \texttt{nil}.

Description

The function \texttt{java-primitive-array-element-type} returns the element type (as a keyword as listed in “Types and conversion between Lisp and Java” on page 166) if \texttt{object} is an array with primitive element type. If \texttt{object} is some other type of Java object, \texttt{java-primitive-array-element-type} returns \texttt{nil}. Otherwise it signals an error.

Notes

1. \texttt{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 \texttt{object} is any array (primitive or not), use \texttt{java-array-element-type} instead. Sometimes \texttt{java-object-array-element-type} may be more convenient.

See also

\texttt{java-array-element-type}

\texttt{java-object-array-element-type}

“Working with Java arrays” on page 178

\textbf{java-serious-exception} \quad \textit{Condition}

Summary

Conditions signaled when something in the system is not really not as it should be.

Package

\texttt{lw-ji}

Superclasses

\texttt{java-exception}

Subclasses

\texttt{java-id-exception}

\texttt{java-low-level-exception}

Description

The condition class \texttt{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-type-to-lisp-array-type  \texttt{jtype} => \texttt{l-result}

lisp-array-type-to-java-type \texttt{lisp-type}

**Arguments**

\texttt{jtype} \hspace{1cm} A foreign type.

\texttt{lisp-type} \hspace{1cm} A Lisp type specifier.

**Values**

\texttt{l-result} \hspace{1cm} A Lisp array element type, or \texttt{nil}.

**Description**

The function \texttt{java-type-to-lisp-array-type} returns the matching Lisp array element type for the foreign type \texttt{jtype}, which needs to be one of the foreign types corresponding to a Java primitive type, or \texttt{nil} if the argument is not such a foreign type.

The function \texttt{lisp-array-type-to-java-type} returns the matching foreign type, corresponding to a Java primitive type, for the Lisp array element type \texttt{lisp-type}, or \texttt{nil} if there is no match.
Both functions use the table below for doing the match:

Table 41.1 Correspondence between Java foreign types and Lisp array element types

<table>
<thead>
<tr>
<th>Foreign type</th>
<th>Lisp type</th>
</tr>
</thead>
<tbody>
<tr>
<td>jbyte</td>
<td>(signed-byte 8)</td>
</tr>
<tr>
<td>jshort</td>
<td>(signed-byte 16)</td>
</tr>
<tr>
<td>jint</td>
<td>(signed-byte 32)</td>
</tr>
<tr>
<td>jlong</td>
<td>(signed-byte 64)</td>
</tr>
<tr>
<td>jdouble</td>
<td>double-float</td>
</tr>
<tr>
<td>jfloat</td>
<td>single-float</td>
</tr>
<tr>
<td>jchar</td>
<td>(unsigned-byte 16)</td>
</tr>
<tr>
<td>jboolean</td>
<td>(unsigned-byte 8)</td>
</tr>
</tbody>
</table>

java-vm-poi

Summary  The Java virtual machine pointer type.

Package   lw-ji

Syntax    java-vm-poi

Description The FLI type java-vm-poi is used for the 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.

Compatibility note At the time of writing, java-vm-poi is not available in LispWorks 7.0.

See also  init-java-interface
### jboolean, jbyte, jchar, jdouble, jfloat, jint, jlong, jshort

**FLI type descriptors**

| Summary | FLI types corresponding to the Java primitive types. |
| Package | lw-ji |
| Description | These FLI types correspond to the Java primitive types. Normally you do not need to use any of these. See “Types and conversion between Lisp and Java” on page 166 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).

jobject 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 jobject 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 jobject. The function jobject-class-name returns the name of the actual class of the jobject, and also caches it in the jobject.

**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 166

**jobject-class-name**

*Function*

**Summary**
Returns the name of the class to which a jobject belongs.
**jobject-class-name**

**Argument**

- **jobject**: A `jobject`.

**Value**

- **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 `jobject` belongs. The name is then cached in the `jobject`.

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**

- `jobject`
- `jobject-of-class-p`
- `jobject-pretty-class-name`
- “Types and conversion between Lisp and Java” on page 166

---

**jobject-ensure-global**

**Function**

**Summary**

Returns a global `jobject` pointing to the same Java object as the argument.

**Argument**

- **jobject**: A `jobject`.

**Value**

- **global-jobject**: A `jobject`.

**Package**

`lw-ji`
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{jobject}s are global anyway. However, when
using \texttt{map-java-object-array}, by default, the \texttt{jobject}s are
local and cannot be used outside the scope of the function
that was passed to \texttt{map-java-object-array}. Similarly,
\texttt{jobject}s 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 func-
tion.

If the argument \texttt{jobject} is not a \texttt{jobject} an error is signaled.

If the argument \texttt{jobject} is already a global reference, \texttt{jobject-}
ensure-global simply returns it.

Notes
1. \texttt{jobject-ensure-global} cannot access local references
   outside the right scope (like any other function).

2. \texttt{jobject-ensure-global} does not accept an instance of
   \texttt{standard-java-object}.

See also
\texttt{jobject-p}
\texttt{map-java-object-array}
\texttt{define-lisp-proxy}

\texttt{jobject-of-class-p} \hspace{1cm} \textit{Function}

Summary
The predicate for whether a Java object is an instance of a
given Java class.

Package \texttt{lw-ji}
### Signature

**jobject-of-class-p**  
`object class-spec => result`

### Arguments

- **object**: A Java object.
- **class-spec**: A class specifier that **find-java-class** accepts or a Java class.

### Values

- **result**: A generalized boolean.

### Description

The function **jobject-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 **find-java-class** accepts, or a Java class, that is a **Object** of class **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 166

---

### **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-object  
“Types and conversion between Lisp and Java” on page 166

---

**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 *jobject* => *name*
```

**Arguments**  
* *jobject*  
  A *jobject*.

**Values**  
* *name*  
  A string.

**Description**  
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 166

---

**jobject-string**  
*Function*

**Summary**  
Calls the Java method `Object.toString` on a Java object.
The LW-JI Package

**Package** lw-ji

**Signature** jobject-string object => string

**Arguments**
- object: 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 `object` on it.

**See also**
- jobject

“Types and conversion between Lisp and Java” on page 166

---

**jobject-to-lisp**

**Function**

**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 jobject or nil.

**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 match-
ing Lisp object. See “Types and conversion between Lisp and Java” on page 166 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 job-object-to-lisp returns nil for failure. When nil-when-fail is false, job-object-to-lisp returns the jobobject 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 (job-object-to-lisp my-obj nil)))
  (if (eq my-obj my-res)
      (fail-branch)
      (success-branch))))

jvref

Summary

Read and set an element in a Java array.

Package

lw-ji

Signature

jvref array index => element

(setf jvref) 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 jobobject 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 166.

(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.
- boolean
  nil or t.
- char
  Integers in the inclusive range \([0, \#xfff]\).

For a non-primitive array, new-value must be convertible to a object 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 166), 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 (\texttt{map-java-object-array}, \texttt{primitive-array-to-lisp-array}, \texttt{lisp-array-to-primitive-array}) can be much faster.

\texttt{jvref} and \texttt{(setf jvref)} access the top level of the array. If \texttt{array} is multi-dimensional, \texttt{jvref} and \texttt{(setf jvref)} will return and set the sub-array. See \texttt{jaref} for accessing elements in a multi-dimensional array.

\texttt{jvref} and \texttt{(setf jvref)} are slightly faster than \texttt{jaref} and \texttt{(setf jaref)} with one index, and give a proper error when called with the wrong number of arguments.

See also

\texttt{map-java-object-array}
\texttt{lisp-array-to-primitive-array}
\texttt{primitive-array-to-lisp-array}
\texttt{jaref}

“Working with Java arrays” on page 178

\textbf{lisp-java-instance-p}

\textit{Function}

\textbf{Summary}
The predicate for objects of type \texttt{standard-java-object}.

\textbf{Package}\lw-ji

\textbf{Signature}\texttt{lisp-java-instance-p object => result}

\textbf{Arguments}\texttt{object} A Lisp object.

\textbf{Values}\texttt{result} A boolean.

\textbf{Description}
The function \texttt{lisp-java-instance-p} is the predicate determining whether an object is an instance of \texttt{standard-java-object}.
The LW-JI Package

See also

jobject-p
jobject
get-jobject
ensure-is-jobject

“Types and conversion between Lisp and Java” on page 166
“CLOS partial integration” on page 185

lisp-to-jobject

Function

Summary
Converts a Lisp object to an appropriate object.

Package
lw-ji

Signature
lisp-to-jobject lisp-object &optional errorp => result

Arguments
lisp-object A Lisp object.

Values
result A jobject or nil.

Description
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 166 for a full description.

If it fails, lisp-to-jobject calls cl:eror, 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 166

make-java-array

Function

Summary
Create a Java array object.
Package    lw-ji
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          Function
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-constructor` 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 185

make-lisp-proxy
make-lisp-proxy-with-overrides

Functions

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 jobject 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 jobject-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 jobject-ensure-global). If the jobject 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 “Overrides” 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 175

**map-java-object-array**

*Function*

**Summary**
Apply a function to the elements in an array.

**Package**
lw-ji

**Signature**
map-java-object-array function array &key collect reverse start end pass-args convert write-back => result

**Arguments**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>A function designator or nil.</td>
</tr>
<tr>
<td>array</td>
<td>A Java array of non-primitive type.</td>
</tr>
<tr>
<td>collect</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>reverse</td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td>start, end</td>
<td>Bounding index designators for array.</td>
</tr>
<tr>
<td>pass-args</td>
<td>One of the keywords :element, :index and :element-index.</td>
</tr>
<tr>
<td>convert</td>
<td>nil, t, or one of the keywords :force-nil, :force-local and :force-global.</td>
</tr>
<tr>
<td>write-back</td>
<td>A generalized boolean.</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>A list or a Lisp vector.</td>
</tr>
</tbody>
</table>
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 `job-object`, 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 objects 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 objects. 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 object (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 objects (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 objects, 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 object 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 objects out of scope, you can convert local objects to global using TObject-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-TObject-array signals an error of type TObject-array-error.

Notes


2. The function TObject-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-TObject-array may be much faster than accessing the elements using jvref or jaref.

4. map-TObject-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
ejaref
primitive-array-to-lisp-array
lisp-array-to-primitive-array
get-primitive-array-region
set-primitive-array-region
tObject-array-element-type
tObject-ensure-global
“Working with Java arrays” on page 178
**Functions**

### primitive-array-to-lisp-array

**Summary**
Copy elements between a Java primitive array and a Lisp array of matching type.

**Package**
lw-ji

**Signature**

```lisp
primitive-array-to-lisp-array p-array &key start end target-start target-end lisp-array => l-result
```

**Signature**

```lisp
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.

The Lisp array that is passed to lisp-array-to-primitive-array must be 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 java-array-element-type)</th>
<th>Lisp type (result of cl:array-element-type)</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>
</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 178

Table 41.2  Correspondence between Java primitive and Lisp array element types

<table>
<thead>
<tr>
<th>Java primitive type</th>
<th>Keyword (result of java-array-element-type)</th>
<th>Lisp type (result of cl:array-element-type)</th>
</tr>
</thead>
<tbody>
<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.
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

Signature
checked-read-java-field full-field-name &optional object =>
field-value-or-nil, nil-or-condition

Signature
set-java-field full-field-name value &optional object => value

Signature
check-java-field full-field-name static-p => result

Signature
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.

Values
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 166) 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-set-
The LW-JI Package

The LW-JI Package is useful for checking whether set-java-field can be used on a field, and whether a value is suitable to be stored in his field, by using object-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  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 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 create-
instance-from-jobject and ensure-lisp-classes-from-tree.

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 standard-java-object).

See also
- write-java-class-definitions-to-stream
- write-java-class-definitions-to-file
- import-java-class-definitions
- create-instance-from-jobject
- ensure-lisp-classes-from-tree

### report-error-to-java-host

**Function**

**Summary**
Tries to report an error to the Java host.

**Package**
lw-ji

**Signature**
report-error-to-java-host error-string log-file-string => result

**Arguments**
- error-string A string.
- log-file-string A string or nil.

**Values**
result A boolean.

**Description**
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 \texttt{init-java-interface} is passed with a function that invokes the user Java error reporters (set in Java by \texttt{com.lispworks.Manager.setErrorReporter} and \texttt{com.lispworks.Manager.setGuiErrorReporter}).

The default function just prints to \texttt{cl:*terminal-io*}, which may be useful enough for debugging.

\texttt{result} is \texttt{t} if there is a function, and \texttt{nil} otherwise.

See also \texttt{init-java-interface}

\begin{itemize}
\item \texttt{reset-java-interface-for-new-jvm}\hfill \textit{Function}
\end{itemize}

\begin{tabular}{ll}
\textbf{Summary} & Resets the Java interface. \\
\textbf{Package} & \texttt{lw-ji} \\
\textbf{Signature} & \texttt{reset-java-interface-for-new-jvm \&key for-shaking-p} \\
\textbf{Arguments} & \texttt{for-shaking-p} \hfill A generalized boolean. \\
\textbf{Values} & None. \\
\textbf{Description} & The function \texttt{reset-java-interface-for-new-jvm} resets the Java interface, which means eliminating all cached information. \texttt{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 \texttt{for-shaking-p} is \texttt{nil}.
\end{tabular}
send-message-to-java-host  

Function

**Summary**
Sends a message to the Java host.

**Package**
lw-ji

**Signature**
send-message-to-java-host message-string where-keyword => result

**Arguments**

*message-string*  
A string.

*where-keyword*  
One of the keywords: **append**, **prepend**, **append-no-scroll** and **reset**.

**Values**

*result*  
A boolean.

**Description**
The function `send-message-to-java-host` sends a message to the Java host. It funcalls the function that was passed as the `send-message-to-java-host` argument to `init-java-interface`, or the default function, with *message-string* and *where-keyword*.

On Android `init-java-interface` is given a function that ends up calling the method `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.

**See also**

`init-java-interface`
`format-to-java-host`

setup-field-accessor  

Function

**Summary**
Defines a Java field accessor.
Package lw-ji

Signature
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.
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

Summary
Define a Java caller, which is a function that calls a Java method or a constructor.

Package
lw-ji

Signature
setup-java-caller name class-name method-name &key signatures => result, condition

Signature
setup-java-constructor name class-name &key class-symbol signatures => result, condition

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.
condition A condition object.

Description
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 `default-compiler-arguments` and `define-java-constructor`.

Unlike the macros `default-compiler-arguments` 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 `default-compiler-arguments
“Defining specific callers” on page 170

**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 :jobject

Accessors java-instance-jobject

Description The class standard-java-object is a class for jobject.

Instances of standard-java-object can be passed to the Java interface functions and callers you define, and returned from Java calls whenever a jobject is needed. Each instance is normally associated with a jobject, which is used by the Java interface.

Apart from accessing the jobject 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 \texttt{jobject} slot defaults to \texttt{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 \texttt{java-instance-jobject}. When you do that, the object must be a \texttt{jobject}, 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 \texttt{jobject} of any Java class.

The other way to set the \texttt{jobject} slot is to use one of the interface functions that does it implicitly. This include the functions \texttt{make-java-instance}, \texttt{create-instance-jobject} and \texttt{create-instance-jobject-list}, and the \texttt{:construct} keyword argument to \texttt{make-instance}.

The \texttt{:construct} keyword is processed by an \texttt{:after} method of \texttt{cl:initialize-instance} on \texttt{standard-java-object}. When \texttt{:construct} is supplied, it needs to be either a list (possibly \texttt{nil}) of arguments for the constructor, or \texttt{t}, which means use \texttt{default-constructor-arguments} to get the argument list. The method then calls \texttt{create-instance-jobject-list} with the instance and the arguments to set create and set the \texttt{jobject}.

Additionally, the instance that is returned by \texttt{create-instance-from-jobject} also has the \texttt{jobject}.

\textbf{Notes}

When you pass \texttt{:construct \texttt{t}}, the call to \texttt{default-constructor-arguments} happens inside \texttt{cl:initialize-instance}, before all the \texttt{cl:initialize-instance} methods were called (the actual order of calls is the standard order). That means that if \texttt{default-constructor-arguments} depends on some values in the instance that may be set by an \texttt{cl:initialize-instance} method of another class, it may not work properly. You can avoid this problem by not pass-
ing 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 175

---

**verify-java-caller**

*Function*

**Summary**
Verify the Java caller.

**Package**
w-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 **default-constructor-arguments** (but not any of the other definers or **setup-java-caller**). If it is not, an error is signaled. Note that the importing interface defines the caller using **default-constructor-arguments** and that **define-java-callers** also expands to **default-constructor-arguments**, 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 default-constructor-arguments explicitly because that does not do any lookup until runtime, 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
default-constructor-arguments
define-java-callers
“Calling from Lisp to Java” on page 168

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 :name-only, :name, :info-only and :successful.
The function `verify-java-callers` verifies all Java callers and returns information about which was successful and which was not.

If `classes` is non-nil, it must be a list of strings specifying Java classes. In this case, `verify-java-callers` verifies only callers for these classes. By default `verify-java-callers` verifies all callers that were defined by `default-constructor-arguments`.

`return` specifies what to return. See below for details.

`verify-java-callers` maps through all the callers that were defined by `default-constructor-arguments` on all classes (if `classes` is `nil`) or on the supplied `classes`.

Note that the importing interface defines the caller using `default-constructor-arguments` and that `define-java-callers` also expands to `default-constructor-arguments`, so `verify-java-callers` verifies these callers too.

`verify-java-callers` does not verify constructors or field accessors.

For each caller, `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.

`verify-java-callers` returns a list containing an item for each failed lookup, except when `return` is the keyword :successful, in which case there is an item for each successful lookup. The value of each item depends on the value of `return` as follows:

- Each item is a cons `(args . condition)` where `args` is a list `:(name class-name method-name)` of the required arguments of the `default-constructor-arguments` form, and `condition` is the condition that was produced.
when looking up. Unless something very unusual happened, this condition will be of type either `java-class-error` (if it failed to find the class) or `java-method-error` (if it failed to find the method).

:name-only Each item is the name of the caller that failed.

:name Each item is a cons where the `cl:car` is the name caller and the `cl:cdr` is the condition that was generated when trying the lookup.

:info-only Each item is the list `(name class-name method-name)` of the required arguments for `default-constructor-arguments` of the failed caller.

:successful Each item is the name of a successful caller.

The default value of `return` is `t`.

`verify-java-callers` requires running Java.

Verification is useful to guard against typing mistakes when you typed the `define-java-callers` explicitly because that does not do any lookup until runtime, 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 `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 `verify-java-caller`
`default-constructor-arguments`
`define-java-callers`

“Calling from Lisp to Java” on page 168
verify-lisp-proxy
verify-lisp-proxies

Functions

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 `do-undefined-method` is non-nil, and requires running Java. The default value of `do-undefined-method` is `nil`.

`verify-lisp-proxy` returns two values:

- `unbounds` reports symbols lacking function definitions. For each list in `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".

- `undefined-methods` (if `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 `method-descs`.

`verify-lisp-proxies` maps through the proxy definitions that were defined by `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 `defs-with-unbounds` is a list corresponding to a definition with symbol not `fbound`, where the `cl:car` is the definition name and the `cl:cdr` is the `unbounds` list as returned by `verify-lisp-proxy`. Each item in `defs-with-undefined-methods` is a cons corresponding to a definition where a method is undefined, where the `cl:car` is the definition name and the `cl:cdr` is a string or a list of undefined methods as described above.

**Notes**

Failure to find an interface is a real error, and will cause `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`

**Signature**

`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 169
41 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

```java
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 7-0-0-0/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`).
Java classes and methods

```java
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 166. 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 166.

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.lisp-works.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 Lisp-Works 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.call-
  DoubleV("log", 3);
=> logThree = 1.0986123
```

**com.lispworks.LispCalls.createLispProxy**

**Method**

```java
public static native Object createLispProxy(String name)
```

**Description**

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.checkLispSymbol

**Method**

```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.

See also

“Calling from Java to Lisp” on page 173
\texttt{init-java-interface}
\texttt{define-lisp-proxy}
\texttt{deliver}
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

```java
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)
```

```java
public static int status()
public static int init_result_code()
public static String mInitErrorString = ""
```

```java
public static boolean loadLibrary()
public static boolean loadLibrary(String deliverName)
```

```java
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
```

```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
```

Error handling

```java
public static void setErrorReporter(LispErrorReporter ller)
public static void setGuiErrorReporter(LispGuiErrorReporter ller)
```

```java
public interface LispErrorReporter
public interface LispGuiErrorReporter
```

```java
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

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 190

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 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 190

com.lispworks.Manager.status  Method and Fields

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 190

**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 (`libs/armeabi-v7a` for Eclipse, `jniLibs/armeabi-v7a` for
Android Studio), or its name is not correct. See `deliver-to-android-project` for details.

**INIT_ERROR_NO_ASSET**

`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 `deliver-to-android-project` for details.

**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 `InitLispWorks`. In general:

- 0 or greater means success (`com.lispworks.Manager.status` returns `STATUS_READY`).
- Values greater than -100 and lower than 0 mean timeout. Since `com.lispworks.Manager.init` is asynchronous, that would be the values during initialization (`com.lispworks.Manager.status` returns `STATUS_INITIALIZING`).
- -100 means not initialized (`com.lispworks.Manager.status` returns `STATUS_NOT_INITIALIZED`).
- Values lower than -100 indicate an error (`com.lispworks.Manager.status` returns `STATUS_ERROR`).

`init_result_code` would typically be used after `com.lispworks.Manager.status` returned `STATUS_ERROR`.

When there is an error, `com.lispworks.Manager.mInitErrorString` contains a string describing it.

See also `com.lispworks.Manager.mInitErrorString`

`com.lispworks.Manager.init`
**Field**

`com.lispworks.Manager.mInitErrorString`

```java
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 190

**Method**

`com.lispworks.Manager.loadLibrary`

```java
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

public interface LispErrorReporter {  
boolean report(String errorString, String filename);  
}

public static void setErrorReporter(LispErrorReporter ler)

public interface LispGuiErrorReporter {  
boolean report(String errorString, String filename);  
}

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

Method

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

Field

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.
public static void showBugFormLogs(Activity act)

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:


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

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

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.

The where argument needs to be one of the four 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_APPEND and ADDMESSAGE_PREPEND mean adding the string in the end or the beginning respectively of the
By default, when adding the string to a textview, addMessage causes it to scroll such that the top of the message is visible. ADDMESSAGE_APPEND_NO_SCROLL does like ADDMESSAGE_APPEND but without ever scrolling. 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).

See also

send-message-to-java-host
com.lispworks.Manager.setErrorReporter
com.lispworks.Manager.setMessageHandler
com.lispworks.Manager.setTextView
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

public static synchronized void setTextView(android.widget.TextView textview)

Description
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 $\text{TextView}$ to another $\text{TextView}$ or to $\text{null}$ without losing text.

The intention is that $\text{TextView}$ 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 $\text{com.lispworks.Manager.setMessageHandler}$.

There is no expectation by $\text{setTextView}$ or $\text{com.lispworks.Manager.addMessage}$ about the properties of the $\text{TextView}$ 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.

$\text{setTextView}$ can be called on any thread, and is thread-safe. The manipulation of the $\text{TextView}$ by $\text{com.lispworks.Manager.addMessage}$ is always done on the GUI process.

See also

$\text{com.lispworks.Manager.addMessage}$
$\text{com.lispworks.Manager.setMessageHandler}$

$\text{com.lispworks.Manager.getClassLoader}$
$\text{com.lispworks.Manager.getApplicationContext}$

Methods

```
public static ClassLoader getClassLoader()
public static Context getApplicationContext()
```

Description

Return the application context of the $\text{context}$ that was supplied to $\text{com.lispworks.Manager.init}$, and the $\text{ClassLoader}$ associated with it.

These are utility methods that LispWorks itself uses and you may find useful. They must be called only after $\text{com.lispworks.Manager.init}$ was called.

See also

$\text{com.lispworks.Manager.init}$
**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
public 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

public class BugFormViewer extends Activity

Description
Used by com.lispworks.Manager.showBugFormLogs to show an individual log.

See also
com.lispworks.Manager.showBugFormLogs
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 **Macro**

**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-
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 UNIX interrupts. 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`.
See also  
current-process-block-interrupts  
current-process-unblock-interrupts  
process-break  
process-interrupt  
process-kill  
process-reset  
process-stop  
with-interrupts-blocked  

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  
barrier-arriver-count
**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**  
For a barrier that is actually in use, the arriver count can change at any time.

**See also**  
`barrier-wait`
`make-barrier`

**barrier-block-and-wait**  
*Function*

**Summary**  
Enables a barrier, waits until a specified number of arrivers arrive, and then wakes immediately.

**Package**  
`mp`

**Signature**  
`barrier-block-and-wait barrier count &key wait-if-used-p errorp timeout unblock => result`

**Arguments**  
`barrier`  
A barrier object.

`count`  
A positive integer.

`wait-if-used-p`  
A generalized boolean.

`errorp`  
A boolean.
timeout  
nil or a positive real.

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 object 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`.

`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 the barrier should be unblocked first. The default value of `unblock` is `nil`.

`barrier-block-and-wait` is "using" the barrier, and only one process can do it the same time. `barrier-block-and-wait` first tries to mark the barrier as used by the current process. It 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. By default, it returns the other process.

2. If `wait-if-used-p` is non-nil, it calls `barrier-wait` on the barrier `barrier` (without any keyword argument) and returns the result.

3. If `errorp` is non-nil, it calls `error`.
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 the barrier is equal or bigger than `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 the 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 intention of `barrier-block-and-wait`. If other processes call functions that manipulate the arriver count or the count (`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 (maybe with 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. The argument 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 will call 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-wait
make-barrier
barrier-enable
barrier-disable
**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 the barrier is enabled and the arriver count is less than `new-count`, this just sets the count of the barrier to the `new-count` and returns t. Otherwise, it calls

`(barrier-unblock barrier :reset-count new-count)`

and returns `nil`.

See also

`barrier-unblock`

**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 for a disabled barrier.

Notes

The count value can be changed by barrier-unblock, barrier-enable, barrier-disable or barrier-change-count.

See also

barrier-wait
make-barrier
barrier-change-count
barrier-disable
barrier-enable
barrier-unblock

barrier-disable

Function

Summary

Unblocks and disables a barrier.

Package

mp

Signature

barrier-disable barrier &optional kill-waiting

Arguments

barrier A barrier.

kill-waiting A boolean.

Description

The function barrier-disable unblocks the barrier barrier and then disables it. If kill-waiting is true, barrier-disable also kills any waiting thread. This is done by calling

(barrier-unblock barrier :disable t :kill-waiting kill-waiting)
See also
barrier-unblock
barrier-wait
make-barrier

barrier-enable

Function

Summary
Ensures that a barrier is enabled.

Package
mp

Signature
barrier-enable barrier count &optional kill-waiting

Arguments
barrier A barrier.
count A positive fixnum, or t meaning most-positive-fixnum.
kill-waiting A boolean.

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-wait
make-barrier
barrier-unblock

barrier-name

Function

Summary
Returns the name of the barrier

Package
mp
Signature  \texttt{barrier-name barrier => name}

Arguments  \texttt{barrier}  A barrier.

Values  \texttt{name}  A string.

Description  The function \texttt{barrier-name} returns the name of the barrier, as supplied or defaulted in the call to \texttt{make-barrier}.

See also  \texttt{barrier-wait}
\texttt{make-barrier}

\textbf{barrier-pass-through}  \textit{Function}

Summary  Increments the arriver count of a barrier.

Package  \texttt{mp}

Signature  \texttt{barrier-pass-through barrier => result}

Arguments  \texttt{barrier}  A barrier.

Values  \texttt{result}  One of the keywords \texttt{:unblocked} and \texttt{:passed-through}.

Description  The function \texttt{barrier-pass-through} increments the arriver count of the barrier \texttt{barrier}. If the arriver count thereby reaches the count, \texttt{barrier-pass-through} unblocks the barrier and returns \texttt{:unblocked}, otherwise it returns \texttt{:passed-through}.

\texttt{barrier-pass-through} is equivalent to calling \texttt{barrier-wait} with \texttt{pass-through} \texttt{t}. See \texttt{barrier-wait} for details.

See also  \texttt{barrier-wait}
\texttt{make-barrier}
barrier-unblock  

**Function**

**Summary**  
Unblocks a barrier.

**Package**  
mp

**Signature**  
barrier-unblock barrier &key disable reset-count kill-waiting

**Arguments**  

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 the barrier, which means that any thread that is waiting on the barrier wakes and returns from `barrier-wait`, and the arriver count is reset to 0.

If `disable` is true, or if `disable` is not passed and the barrier was made with `disable-on-unblock` true, then `barrier-unblock` also disables the barrier, so any further calls to `barrier-wait` return `nil` immediately.

If `reset-count` is true, it must be valid count (a positive fixnum or t), and `barrier-unblock` sets the count of the 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-wait  
make-barrier
**barrier-wait**  

Function

**Summary**  
Waits on a barrier until enough threads arrive.

**Package**  
mp

**Signature**  

\[
\text{barrier-wait} \quad \text{barrier} \quad \&\text{key} \quad \text{timeout} \quad \text{callback} \quad \text{pass-through} \\
\quad \text{discount-on-abort} \quad \text{discount-on-timeout} \quad \text{disable-on-unblock} \quad \Rightarrow \quad \text{result}
\]

**Arguments**

- **barrier**: A barrier.
- **timeout**: A non-negative number.
- **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. 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, it does not actually wait.

**discount-on-abort** controls whether to change the arrivers count on an abort.
*discount-on-timeout* controls whether to change the arrivers count on a timeout.

*disable-on-unblock* controls whether to disable the barrier when unblocking.

*callback*, if supplied, specifies a callback called before unblocking.

*barrier-wait* first checks whether the barrier is disabled, and if it is *barrier-wait* returns *nil* immediately. It then checks the number of arrivers, 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, and then waits for the barrier to be unblocked (unless *pass-through* is true). If the number of arrivers is the count minus 1, *barrier-wait* unblocks the barrier (described below) and returns *:unblocked*.

*discount-on-abort*, *discount-on-timeout*, *disable-on-unblock* and *callback* allow you to control the waiting and also the unblocking of the barrier. 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 the 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*, meaning no timeout. 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 after a timeout *barrier-wait* decrements the arrivers count, 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-inter-
rupt), by default it does not change the arrivers count, 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 on aborting barrier-wait decrements the arrivers count, 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 the barrier: when the number of arrivers is the count of the barrier minus 1, barrier-wait "unblocks the barrier". This involves the following steps:

1. If callback is true it is called with the barrier while holding an internal lock on the barrier. See the comment in make-barrier. If the callback aborts, nothing has been changed in the barrier (including no change to the arrivers).

2. The barrier is marked as unblocked for the currently waiting threads.

3. The number of arrivers in the barrier is reset to 0. Unless the next step disables the barrier, this means that any subsequent call to barrier-wait will wait, as if the barrier had just been created.

4. If disable-on-unblock is true, barrier-wait then disables the barrier. That means that until it is enabled, any call to barrier-wait will return immediately.

5. It wakes up all the waiting threads.

6. It returns the symbol :unblocked.

The possible values of result occur in these circumstances:

- The current process waited and some other process unblocked the barrier.
The current process unblocked the barrier.

The wait timed out.

Pass through because pass-through was true.

The barrier is disabled.

See also

- barrier-arriver-count
- barrier-block-and-wait
- barrier-change-count
- barrier-count
- barrier-disable
- barrier-enable
- barrier-name
- barrier-pass-through
- barrier-unblock
- make-barrier

---

cancel

Function:

change-process-priority

| Summary | Changes the priority of a process. |
| Package | mp |
| Signature | change-process-priority process new-priority => new-priority |
| Arguments | process A process. |
| | new-priority A fixnum. |
| Description | Changes the priority of process to be new-priority. |
| See also | process-priority |
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 this condition variable, 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-variable-signal

Function

Summary
Wakes one thread waiting on a given condition variable.

Package
mp
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 this condition variable, but this is not required. When using the lock, you may prefer to use `lock-and-condition-variable-signal`.

The return value `signaledp` is non-nil if a process was signaled, or `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-variable-wait

**Function**

**Summary**

Waits for a given condition variable to be signaled.

**Package**

`mp`

**Signature**

```
condition-variable-wait condvar lock &key timeout wait-reason => wakep
```

**Arguments**

- `condvar` A condition variable
- `lock` A `mp:lock`
The function `condition-variable-wait` waits at most `timeout` seconds for the condition variable `condvar` to be signaled. The lock `lock` is released while waiting and claimed again before returning. The caller must be holding the lock `lock` before calling this function.

The return value `wakep` is non-nil if the signal was received or `nil` if there was a timeout. If `timeout` is `nil`, `condition-variable-wait` waits indefinitely.

If `wait-reason` is non-nil, it is used as the `wait-reason` while waiting for the signal.

It is recommended that you use `lock-and-condition-variable-wait` or `simple-lock-and-condition-variable-wait` instead of `condition-variable-wait`. The locking functions make it easier to avoid mistakes, and can be more efficient.

`timeout` controls how long to wait for the signal: before returning, the function waits to claim the lock, possibly indefinitely.

See also
- `condition-variable-wait-count`
- `make-condition-variable`
- `lock-and-condition-variable-wait`
- `simple-lock-and-condition-variable-wait`
- `lock-and-condition-variable-signal`
- `lock-and-condition-variable-broadcast`
- `condition-variable-signal`
- `condition-variable-broadcast`
**condition-variable-wait-count**

*Function*

Summary: Returns the current number of threads that are still waiting for the condition variable.

Package: `mp`

Signature: `condition-variable-wait-count condvar => wait-count`

Arguments: `condvar` A 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 the condition variable. Note that for a condition variable that is actually in use, this number can change at any time.

See also: `condition-variable-wait`

*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.
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 UNIX interrupts. 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  

Function

Summary  
The predicate for whether the current process is cleaning up after being killed.

Package  
mp

Signature  
current-process-in-cleanup-p => result

Values  
result  
A boolean.

Description  
The function current-process-in-cleanup-p returns true after the current process is killed. In particular, it returns true while the cleanups that were set by ensure-process-cleanup execute.

See also  
ensure-process-cleanup

current-process-kill  

Function

Summary  
Kill the current process.

Package  
mp

Signature  
current-process-kill

Arguments  
None.

Values  
None.

Description  
The function current-process-kill kills the current process.

current-process-kill signals an error if it is called when interrupts are blocked, unless it is inside the scope of with-other-threads-disabled, in which case the process is
marked as "dying", and actually dies on exit from with-other-threads-disabled.

Normally, current-process-kill throws out and does not return. It does execute all surrounding unwind-protect forms.

If current-process-kill is called while the process is already doing cleanups, it just returns.

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 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 with-other-threads-disabled

current-process-pause

Function

Summary
Sleeps for a specified time, but can be woken up.

Package
mp

Signature
current-process-pause time &optional function &rest args => result

Arguments
time
A positive number.

function
A function designator.

args
Arguments passed to function.

Values
The keyword :poked, or nil.

Description
The function current-process-pause sleeps for time seconds, but wakes up if another process did something to wake up the current process (normally this is process-poke, but it
can also be `process-interrupt`, `process-stop`, `process-unstop` or `process-kill`.

`current-process-pause` is quite similar to `cl:sleep`, but it returns if anything causes the process to wake up, even if the time did not pass.

If `function` is passed just before going to sleep, `current-process-pause` applies `function` to `args`, and if this returns a true value `current-process-pause` returns it immediately. 

`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:

```lisp
(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:

```lisp
(loop
  (mp:current-process-pause 60 'check-for-need-cleanup)
  (do-cleanup))
```

Another process which wants to provoke a cleanup will do:

```lisp
(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.
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 238

---

**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 process-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
debug-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 \texttt{debug-other-process} to try to find out why it hangs.

If \texttt{process} is a string, the process is found as if by \texttt{find-process-from-name}. The list of process names can be found via \texttt{ps}.

\textbf{See also} \hfill \texttt{find-process-from-name} \hfill \texttt{process-break} \hfill \texttt{ps}

\textbf{*default-process-priority*} \hfill \textit{Variable}

\textbf{Summary} \hfill The default priority for processes.

\textbf{Package} \hfill \texttt{mp}

\textbf{Description} \hfill The variable \texttt{*default-process-priority*} contains the default priority for processes.

\textbf{See also} \hfill \texttt{process-run-function}

\textbf{ensure-process-cleanup} \hfill \textit{Function}

\textbf{Summary} \hfill Run forms when a given process terminates.

\textbf{Package} \hfill \texttt{mp}

\textbf{Signature} \hfill \texttt{ensure-process-cleanup cleanup-form &key priority force process =>}

\textbf{Signature} \hfill \texttt{ensure-process-cleanup cleanup-form &optional process =>}

\textbf{Arguments} \hfill \texttt{cleanup-form} \hfill Form to run when \texttt{process} terminates.
**priority**  
An integer in the inclusive range \([-1000000, 1000000]\).

**force**  
A boolean.

**process**  
An \texttt{mp:process} object.

**Values**  
None.

**Description**  
The function \texttt{ensure-process-cleanup} ensures that the cleanup-form is present for the process \texttt{process}. When \texttt{process} terminates, its cleanup forms are run. Cleanup forms can be functions of one argument (the \texttt{process}), or lists, in which case the \texttt{cl:car} is applied to the \texttt{process} and the \texttt{cl:cdr} of the list.

\texttt{process} is the process to watch for termination. By default, this is the value returned by \texttt{get-current-process}.

**priority** determines the execution order of the forms. Higher \texttt{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 \texttt{priority} is 0.

**force** determines what to do if the same cleanup is already registered but with a different \texttt{priority}. When adding cleanup forms, \texttt{ensure-process-cleanup} uses \texttt{cl:equal} to ensure that the form is only added once. If a cleanup already exists with the same priority, \texttt{ensure-process-cleanup} just returns \texttt{nil}, otherwise it acts according to \texttt{force}: if \texttt{force} is \texttt{nil} it invokes an error, but if \texttt{force} is \texttt{t} then \texttt{ensure-process-cleanup} removes the old entry before adding the new entry. The default value of \texttt{force} is \texttt{nil}.

**Notes**
1. You can test for whether the current process is executing its cleanups with \texttt{current-process-in-cleanup-p}.
2. For compatibility with LispWorks 6.1 and earlier versions, \texttt{ensure-process-cleanup} can also be called like this:

\[
(\texttt{ensure-process-cleanup cleanup-form process})
\]
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`. If there is no such process, the function returns `nil`.

---

1235
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. Two Background Execute processes exist by default. This is adequate because it is assumed that you normally use process-run-function. If you use funcall-async and/or funcall-async-list often, you probably need to increase the number of these Background Execute processes. You do that by 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 in general, but if you use a wait function and want another process to poke (by process-poke), you can use get-current-process to find the process to poke.

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 238

get-current-process  \textit{Function}

Summary  Returns the current Lisp process.

Package  mp

Signature  get-current-process => result

Values  result  A \texttt{mp:process}, or nil.

Description  The function get-current-process returns the actual process in which it is called. In this respect it differs from \texttt{*current-process*}, which can be bound to another process. In particular, when a process A calls the \texttt{wait-function} of process B, in the \texttt{wait-function} get-current-process returns the process A, but \texttt{*current-process*} is bound to process B. result is nil if multiprocessing is off.

See also  \texttt{*current-process*}
get-process  

**Function**

**Summary**
Returns a process corresponding to a supplied designator.

**Package**
mp

**Signature**
get-process  process-designator  =>  process

**Arguments**

- process-designator

  - A mp:process, a string, a stack-group, a function, a symbol or a fixnum.

**Values**

- process

  - A mp:process.

**Description**

The function get-process returns a process according to the supplied process-designator, which is interpreted as follows:

- mp:process: Return it.

- A string: Find the first process (highest priority) with matching name. Process names are compared by string=.

- A stack-group: Return the process of the stack-group.

- A function: Return the first process that has process-designator as its function (that is, the third argument of process-run-function).

- 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`.

**See also**
- `process-private-property`
- `remove-process-private-property`
- `pushnew-to-process-private-property`
- `remove-from-process-private-property`

**initialize-multiprocessing**

**Function**

**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 and Mac OS X (using the Cocoa image), the LispWorks IDE starts up by default.

See also  

`*initial-processes*`

`process-run-function`
*initial-processes*  

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:

```lisp
(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

Function

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.
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-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

Description

The function lock-and-condition-variable-broadcast locks the lock lock, applies the function setup-function, calls condition-variable-broadcast and unlocks. lock-and-condition-variable-broadcast makes it easier to avoid mistakes in using a condition variable.

lock-and-condition-variable-broadcast performs the equivalent of:
(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-signal
simple-lock-and-condition-variable-signal
lock-and-condition-variable-signal
condition-variable-signal
condition-variable-signal
condition-variable-broadcast
processes-count

lock-and-condition-variable-signal

Function

Summary  Locks, applies a setup function, calls condition-variable-signal and unlocks.

Package  mp

Signature  lock-and-condition-variable-signal lock condvar lock-timeout setup-function &rest args

Description  The function lock-and-condition-variable-signal locks the lock lock, applies the setup-function, calls condition-variable-signal and unlocks. lock-and-condition-variable-signal makes it easier to avoid mistakes in using a condition variable.

lock-and-condition-variable-signal performs the equivalent of:
(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 the 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

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

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 the function predicate to the arguments args. If this call returns nil it performs the equivalent of a call to condition-variable-wait, passing it the 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, the 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 the predicate if this is not nil, otherwise it returns the result of the equivalent call to condition-variable-wait.

Notes

1. predicate and the return-function are called with the 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 errors 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 (condition-variable-signal, condition-variable-broadcast, lock-and-condition-variable-signal, lock-and-condition-variable-broadcast) can be used to wake a process
waiting in lock-and-condition-variable-wait. The non-locking one can be used without the lock when it is useful.

See also
- condition-variable-wait
- simple-lock-and-condition-variable-wait
- lock-and-condition-variable-signal
- lock-and-condition-variable-broadcast
- condition-variable-signal
- condition-variable-broadcast

**lock-locked-p**

Function

Summary
The predicate for whether a lock is locked.

Package
mp

Signature
lock-locked-p lock => result

Arguments
lock A lock.

Values
result A boolean.

Description
The function lock-locked-p is the predicate for whether a 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 the lock is a "sharing" lock, this checks for an exclusive lock.

See also
make-lock

**lock-owned-by-current-process-p**

Function

Summary
Checks whether a lock is locked by the current thread.
### lock-owned-by-current-process-p

**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 the lock is either unlocked or locked by another process.

If the 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**  
`make-lock`

---

### 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 object.

**Values**  
result  
A boolean.

**Description**  
The function `lock-recursive-p` is the predicate for whether the lock `lock` allows recursive locking (that is, whether it can be repeatedly locked by the same process).

See the `make-lock` argument `recursivep`. 
lock-recursively-locked-p

**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 a 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 the lock is a "sharing" lock, lock-recursively-locked-p checks for an exclusive lock.

**See also**

make-lock

lock-name

**Summary**

Returns the name of a lock.

**Package**

mp
**Signature**

lock-name lock => name

**Arguments**

lock A lock object

**Values**

name A string

**Description**

The function lock-name takes a lock object as its argument and returns the name of the lock object.

**Example**

(let ((lock (mp:make-lock :name "my lock")))
  (mp:lock-name lock))

=> "my lock"

**See also**

make-lock
with-lock
process-lock
process-unlock
lock-owner

---

### lock-owner

**Function**

**Summary**

Returns the owner of a lock.

**Package**

mp

**Signature**

lock-owner lock => result

**Arguments**

lock A lock object

**Values**

result A process, t or :unknown

**Description**

The function lock-owner returns the process that currently owns the lock, or nil.

If lock is a "sharing" lock then lock-owner checks for an exclusive lock (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 process (for example, the process is killed while holding the 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-owned-by-current-process-p`
- `make-lock`
- `with-lock`
- `process-lock`
- `process-unlock`
- `lock-name`
- `lock-owned-by-current-process-p`

### mailbox-count

**Function**

**Summary**

Returns the number of objects currently in a mailbox.

**Package**

`mp`

**Signature**

`mailbox-count mailbox => count`

**Arguments**

- `mailbox` A mailbox.

**Values**

- `count` A non-negative integer.
The function `mailbox-count` returns the number of objects currently in the mailbox `mailbox`.

`mailbox` should be an object of type `mp:mailbox`.

A mailbox is empty if its `count` is 0.

See also `mailbox-empty-p`  
`mailbox-not-empty-p`  
`make-mailbox`

---

### `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.

See also `mailbox-not-empty-p`  
`mailbox-send`  
`mailbox-peek`  
`mailbox-read`  
`make-mailbox`

---

### `mailbox-not-empty-p`  
**Function**

**Summary** Tests whether a mailbox has contents.
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**

Returns the first object in a mailbox.

**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`.

**Description**

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`

---

**mailbox-read**

*Function*

Reads the next object in a mailbox.

**Summary**

Reads the next object in a mailbox.

**Package**

`mp`

**Signature**

`mailbox-read mailbox &optional wait-reason timeout => object, flag`

**Arguments**

- `mailbox` A mailbox.
- `wait-reason` A string or `nil`.
- `timeout` A non-negative number 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 number, 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`

---

**mailbox-reader-process**

**Function**

**Summary**

Returns the reader process of a mailbox.

**Package**

`mp`

**Signature**

`mailbox-reader-process mailbox => process`

**Arguments**

`mailbox`  
A mailbox.

**Values**

`process`  
A process or `nil`. 
Description  The function `mailbox-reader-process` returns the reader process of `mailbox`.

**mailbox-send**  

*Function*

**Summary**  Sends 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.

**See also**  `mailbox-empty-p`  
`mailbox-peek`  
`mailbox-read`  
`make-mailbox`

**mailbox-wait**  

*Function*

**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 number or `nil`.

1257
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 number, 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 ware 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 does not add work to the scheduler.

See also

mailbox-not-empty-p
mailbox-empty-p
mailbox-peek
**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.

**Values**

- `result` An event or `nil`.

**Description**

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 it is called as a Process Wait function (see “Generic Process Wait functions” on page 236) 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
**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`.

The 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 the 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 the barrier. It is called with a single argument (the barrier) while holding an internal lock on the barrier that will prevent other operations on the 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 the 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-arriver-count`
`barrier-block-and-wait`
`barrier-change-count`
`barrier-count`
`barrier-disable`
`barrier-enable`
`barrier-name`
`barrier-pass-through`
`barrier-unblock`
`barrier-wait`

---

**make-condition-variable**

**Function**

**Summary**

Makes a condition variable.
make-condition-variable

**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-wait`
- `condition-variable-signal`
- `condition-variable-broadcast`

---

**make-lock**

**Function**

**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.
The function \texttt{make-lock} creates a lock object. See “Locks” on page 232 for a general description of locks.

\texttt{name} names the lock and can be queried with \texttt{lock-name}. The default value of \texttt{name} is "Anon".

\texttt{important-p} controls whether the lock is automatically freed when the holder process finishes. When \texttt{important-p} is true, the system notes that this lock is important, and automatically frees it when the holder process finishes. \texttt{important-p} should be \texttt{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 \texttt{important-p} is \texttt{nil}.

\texttt{safep} controls whether the lock is safe. A safe lock gives an error if \texttt{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 \texttt{safep} is \texttt{t}.

\texttt{recursivep}, when true, allows the lock to be locked recursively. Trying to lock a lock that is already locked by the current thread just increments its lock count. If the lock is created with \texttt{recursivep nil} then trying to lock again causes an error. This is useful for debugging code where the lock is never expected to be claimed recursively. The default value of \texttt{recursivep} is \texttt{t}.

\texttt{sharing}, when true, causes \texttt{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

```lisp
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>
```

See also

*current-process*
lock-recursive-p
process-lock
process-unlock
schedule-timer
with-lock

---

**make-mailbox**

*Function*

**Summary**
Makes a new mailbox.

**Package**
mp

**Signature**

```
make-mailbox &key size name => mailbox
```

**Arguments**

- `size` An integer
- `name` A Lisp object
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-empty-p`

- `mailbox-peek`
- `mailbox-read`
- `mailbox-send`
- `process-whostate`
- `make-unlocked-queue`

---

### 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

```lisp
(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-acquire
- semaphore-count
- semaphore-name
- semaphore-release
- semaphore-wait-count

### make-timer

**Function**

**Summary**

Creates and returns an unnamed timer.

**Package**

mp

**Signature**

`make-timer function &rest arguments => timer`

**Arguments**

- `function` A function
- `arguments` A set of arguments to `function`

**Values**

- `timer` A timer

**Description**

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.

Note that the function `make-named-timer` creates a named timer.

**Example**

```lisp
(setq timer
  (mp:make-timer 'print 10 *standard-output*))
 =>
#<Time Event : PRINT>
```
map-all-processes

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-processes continues by calling function on the next process, but if function returns true then map-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.

**Package**

*mp*

**Signature**

`map-all-processes-backtrace &optional function`

**Arguments**

- `function` A function taking one argument

**Values**

None.

**Description**

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 \( p \) immediately and stops calling \textit{function} (so \textit{function} may not get called on all processes).

\textbf{See also} \texttt{map-all-processes}

\textbf{notice-fd} \hspace{1cm} \textit{Function}

\begin{description}
\item[Summary] Add a file descriptor to the set of interesting input file descriptors.
\item[Package] \texttt{mp}
\item[Signature] \texttt{notice-fd} \( fd \)
\item[Arguments] \( fd \) \quad A UNIX file descriptor
\item[Values] None.
\item[Description] The function \texttt{notice-fd} adds the given \textit{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[See also] \texttt{unnotice-fd}
\end{description}

\textbf{process-alive-p} \hspace{1cm} \textit{Function}

\begin{description}
\item[Summary] Determines if a process is alive.
\item[Package] \texttt{mp}
\item[Signature] \texttt{process-alive-p} \( \text{process} \Rightarrow \text{bool} \)
\item[Arguments] \( \text{process} \) \quad 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 | `(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 `(setf 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**

```lisp
process-exclusive-lock sharing-lock &optional whostate timeout
```

**Arguments**

- `sharing-lock` A sharing lock.
- `whostate` The status of the process while the lock is locked, as seen in the Process Browser.
- `timeout` A timeout interval, in seconds.

**Description**

The function `process-exclusive-lock` is the same as `process-lock`, but on a "sharing" lock. It waits until the lock is free before locking 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 process, but may throw before locking, as described in
“Guarantees and limitations when locking and unlocking” on page 233.

See also

make-lock
process-lock
with-exclusive-lock

**process-exclusive-unlock**

*Function*

**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 the lock, but is not guaranteed to return, as described in “Guarantees and limitations when locking and unlocking” on page 233.

See also

`process-exclusive-lock`
`process-unlock`
`with-exclusive-lock`
**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.

Examples: This example shows a typical use with a bound symbol:

```
(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.

1279
**The MP Package**

**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`

---

**Function**

**process-interrupt**

**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:

\[
\begin{align*}
  (\text{let } ((\text{message } (\text{read-message})))) \\
  (\text{process-message message})
\end{align*}
\]

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 \texttt{process-interrupt} except that the arguments are supplied as a list.

\textbf{See also} \texttt{process-interrupt}

\textbf{process-join} \quad \textit{Function}

\textbf{Summary} Waits until a specified process dies, or a \textit{timeout} is reached.

\textbf{Package} \texttt{mp}

\textbf{Signatures} \texttt{process-join process \&key \ timeout \=> flag}

\textbf{Arguments} \begin{itemize} 
  \item \texttt{process} A process.
  \item \texttt{timeout} A non-negative number.
\end{itemize}

\textbf{Values} \begin{itemize} 
  \item \texttt{flag} A boolean.
\end{itemize}

\textbf{Description} The function \texttt{process-join} waits until the process \texttt{process} dies, or \texttt{timeout} seconds passed.

If the process dies then \texttt{process-join} returns \texttt{t}. If the timeout passed it returns \texttt{nil}.

\texttt{process-join} can be used on dead processes, and in this case returns \texttt{t} immediately.

The effect of \texttt{process-join} is similar to

\texttt{(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 \texttt{process-join} returns immediately when the process dies.

\textbf{See also} \texttt{process-wait-with-timeout}
**process-kill**

*Function*

**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. `process-kill` is deprecated. Use `process-terminate` instead.

**Notes**
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**  
Claims the lock for the current process.

**Package**  
mp

**Signature**  

process-lock lock &optional whostate timeout => result

**Arguments**  

lock  
A lock object (see make-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 not given, process-lock waits until the lock can be set by the current Lisp process. A process can set 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 the owner of the lock is the value of *current-process*, then lock remains locked and an internal count is incremented. The Lisp process sleeps until the lock is claimed or the timeout period 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 233.
Example

(process-lock *my-lock* "waiting to lock" 10)

See also

make-lock
process-exclusive-lock
process-unlock
with-lock

process-mailbox

Function

Summary
Accesses the mailbox associated with a process.

Package
mp

Signature
process-mailbox process => result
(setq process-mailbox) result process => result

Arguments
process A process.

Values
result A mailbox object, or nil.

Description
process-mailbox is an accessor function which returns or sets the mailbox associated with process.

Example
(setq (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

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:

```lisp
(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

(setf 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.

(setf 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 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**
`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 `process-private-property`.

**Example**
`(process-property ‘foo (get-current-process) ‘bar)`
The MP Package

=> BAR
(setf (process-property 'foo) 'foo-value)
=> FOO-VALUE
(process-property 'foo)
=> FOO-VALUE

See also
process-private-property
remove-process-property
remove-from-process-property
pushnew-to-process-property

process-reset

Function

Summary
Resets a process by discarding its current state.

Package
mp

Signature
process-reset process =>

Arguments
process A process.

Values
None.

Description
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.

process-reset modifies the dynamic execution state of process. It performs a non-local exit from the currently running function, to cause the process's main function to return. unwind-protect forms will be run.

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 `process-reset` causes an asynchronous non-local exit, it is possible that it can occur within an `unwind-protect` cleanup form or before data used by an `unwind-protect` cleanup form has been initialized. In some cases, not all cleanups within that form will be run.

**process-run-function**  

*Function*

**Summary**
Create a new process, passing it a function to run.

**Package**  
`mp`

**Signature**

```
process-run-function name keywords function &rest arguments  
=> process
```

**Arguments**

- `name`  
  A name for the new process.

- `keywords`  
  Keywords specifying properties of the new process.

- `function`  
  A function to apply.

- `arguments`  
  Arguments to pass to `function`.

**Values**

- `process`  
  The newly created process.

**Description**

This function creates a new Lisp process with name `name`. Other properties of `process` may be specified in keyword/value pairs in `keywords`:

- `:priority`  
  A `fixnum` representing the priority for the process. If `:priority` is not supplied, the process priority becomes the value of the variable `*default-process-priority*`.

- `:mailbox`  
  A mailbox object, a string, `t` or `nil`, used to initialize the `process-mailbox` of `process`.  

---

1293
True values specify that \textit{process} should have a mailbox. A mailbox object is used as-is; a string is used as the name of a new mailbox; and \texttt{t} causes it to create a mailbox with the same name as \textit{process}, that is, \texttt{name}.

Note that both \texttt{process-send} and \texttt{process-wait-for-event} force the relevant process to have a mailbox.

\texttt{: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".

\texttt{:remote-terminator}

A function designator for a function of one argument.

\texttt{:local-terminator}

A function designator for a function of no arguments.

\texttt{:terminate-by-send}

A generalized boolean.

The new process is preset to apply \textit{function} to \textit{arguments} and runs in parallel, while \texttt{process-run-function} returns immediately.

\texttt{:remote-terminator}, \texttt{:local-terminator} or \texttt{:terminate-by-send} define the Terminate Method of the process, which is what \texttt{process-terminate} uses. If more than one of these keyword arguments is supplied, then \texttt{: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.

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

```lisp
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
Signature

`process-run-reasons` `process => reasons`

`(setf process-run-reasons)` `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

`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 =>

**Arguments**
process A process
object An object
change-priority A fixnum, nil, t, or :default

**Values**
None.

**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.

See also  
general-handle-event  
mailbox-send  
process-wait-for-event  
“Communication between processes and synchronization” on page 238

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 the lock is locked, as seen in the Process Browser.

timeout  
A timeout period, in seconds.

Description  
This is like process-lock, but the lock must be "sharing" and the lock will be locked in shared mode. That means that other threads can also lock it in shared mode.
Before locking this waits for the lock to be free of any exclusive lock, but it does not check for other shared mode use of the same lock.

Calls to \texttt{process-sharing-lock} should be matched by calls to \texttt{process-sharing-unlock}. Normally \texttt{with-sharing-lock} is the best way to achieve this.

\textbf{Notes}

It is possible to lock for sharing inside the scope of sharing lock and inside the scope of exclusive lock.

\texttt{process-sharing-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 233.

\textbf{See also}

\texttt{process-lock}
\texttt{process-sharing-unlock}
\texttt{with-sharing-lock}

\textbf{Function}

\textbf{process-sharing-unlock}

\textbf{Summary}

Removes a sharing lock.

\textbf{Package}

\texttt{mp}

\textbf{Signature}

\texttt{process-sharing-unlock sharing-lock}

\textbf{Arguments}

\texttt{sharing-lock} A sharing lock.

\textbf{Description}

The function \texttt{process-sharing-unlock} is the same as \texttt{process-unlock} but for a "sharing" lock.

Calls to \texttt{process-sharing-unlock} should be matched by calls to \texttt{process-sharing-lock}. Normally \texttt{with-sharing-lock} is the best way to achieve this.
Notes

`process-sharing-unlock` is guaranteed to successfully unlock the lock, but is not guaranteed to return, as described in “Guarantees and limitations when locking and unlocking” on page 233.

See also

`process-unlock`  
`with-sharing-lock`

---

**process-stop**

*Function*

**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 251. `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  

\( \text{mp} \)

Signature  

\( \text{process-terminate} \ \text{process \ &key \ join-timeout \ force-timeout} \)

Arguments  

\( \text{process} \quad \text{A mp:process object.} \)

\( \text{join-timeout} \quad \text{A real number or nil.} \)

\( \text{force-timeout} \quad \text{A real number or nil.} \)

Description  

The function \( \text{process-terminate} \) terminates the process \( \text{process} \), which means killing it "nicely". \( \text{process-terminate} \) invokes the Terminate Method of \( \text{process} \), if it has one, otherwise it calls \( \text{process-kill} \).

The terminate is set either by supplying one of \textit{local-terminator}, \textit{remote-terminator} or \textit{terminate-by-send} in the call to \( \text{process-run-function} \), or by a call to \( \text{current-process-set-terminate-method} \) on the process. See the entry for \( \text{process-run-function} \) for details.

If the process does not have a Terminate Method, \( \text{process-terminate} \) calls \( \text{process-kill} \).

If \( \text{force-timeout} \) is a number then \( \text{process-terminate} \) sets a timer that kills the process after \( \text{force-timeout} \) seconds.

If \( \text{join-timeout} \) is a number then it is the time in seconds to "JOIN" the process, that is waiting for it to die. When \( \text{join-timeout} \) is supplied, after invoking the Terminate Method or calling \( \text{process-kill} \), \( \text{process-terminate} \) calls \( \text{process-join} \) using \( \text{join-timeout} \) as the timeout, and returns the result.

\( \text{process-terminate} \) returns the result of \( \text{process-join} \) if \( \text{join-timeout} \) is non-nil, otherwise it returns 0.

Notes  

1. \( \text{process-terminate} \) is the appropriate way to kill processes, because it gives the process the option to decide when to exit. \( \text{process-kill} \) 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**
Relinquishes a lock held by the current process.

**Package**
`mp`

**Signature**
`process-unlock lock &optional errorp => result`

**Arguments**
- `lock` The lock to be relinquished.
- `errorp` When this is `t`, an error is signaled if `*current-process*` is not the owner of the lock. The default is `t`.

**Values**
- `result` A boolean.

**Description**
Attempts to release a lock. If the lock is owned by `*current-process*`, `process-unlock` decrements an internal count. If this lock count is then zero, the lock is released. 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 the lock, but is not guaranteed to return, as described in “Guarantees and limitations when locking and unlocking” on page 233.

See also

make-lock
process-exclusive-unlock
process-lock
with-lock

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 251.
See also

process-stop
process-stopped-p

process-wait 

Function

Summary
Suspends the current process until a condition is true.

Package
mp

Signature
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 235

### 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.

Description

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 236) 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`.


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 \texttt{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 \texttt{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.

\texttt{process-wait-local} always returns \texttt{t}. 

See also  
process-poke  
process-wait-local-with-periodic-checks  
process-wait-local-with-timeout  
“Process Waiting and communication between processes” on page 235

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-checkswait-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 millisecond-
onds. 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 235
**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 number.

- `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 235
process-wait-local-with-timeout-and-periodic-checks \hspace{1cm} Function

Summary \hspace{1cm} Like \texttt{process-wait-local-with-timeout}, but also calls the wait function periodically.

Package \hspace{1cm} \texttt{mp}

Signature \hspace{1cm} \texttt{process-wait-local-with-timeout-and-periodic-checks (wait-reason timeout period function \&rest args)}

Arguments \hspace{1cm} \begin{itemize}
  \item \texttt{wait-reason} \hspace{0.5cm} A string.
  \item \texttt{timeout} \hspace{0.5cm} A non-negative number.
  \item \texttt{period} \hspace{0.5cm} A positive real number.
  \item \texttt{function} \hspace{0.5cm} A function designator.
  \item \texttt{args} \hspace{0.5cm} Arguments passed to \texttt{function}.
\end{itemize}

Description \hspace{1cm} The function \texttt{process-wait-local-with-timeout-and-periodic-checks} is like \texttt{process-wait-local-with-timeout}, but also calls the wait function periodically.

The \texttt{timeout} and \texttt{period} are both in seconds.

For information about the periodic calls, see \texttt{process-wait-local-with-periodic-checks}.

See also \hspace{1cm} \begin{itemize}
  \item \texttt{process-poke}
  \item \texttt{process-wait-local-with-periodic-checks}
  \item \texttt{process-wait-local-with-timeout}
  \item “Process Waiting and communication between processes” on page 235
\end{itemize}

process-wait-with-timeout \hspace{1cm} Function

Summary \hspace{1cm} 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 timeout, in seconds.
- `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 235
**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.
The function `processes-count` returns the number of Lisp processes that are currently alive.

The `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 `processes-count` was called and the time it returns.

See also `list-all-processes`

**pushnew-to-process-private-property**

Function

Summary

Pushes a new value to a private property of the current process.

Package `mp`

Signature

```
pushnew-to-process-private-property indicator value &key test => result
```

Arguments

- `indicator` A Lisp object.
- `value` A Lisp object.
- `test` A function designator for a function of two arguments.
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 #'eq1.

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

Function

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**  
```lisp
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**  
```lisp
remove-from-process-property indicator value &key process test => result
```

**Arguments**
- indicator  
  A Lisp object.
- value  
  A Lisp object.
A process, or **nil**.

**test**  
A function designator for a function of two arguments.

**result**  
A list.

The function **remove-from-process-property** removes value from the value of the property associated with indicator for the process process. It uses the function test to compare value with existing values, in the same way as *cl:remove*. 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.

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

**Function**

**remove-process-private-property**

**Summary**  
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`

---

**Function**

**remove-process-property**

**Summary**

Removes a general property from a process.

**Package**

`mp`

**Signature**

`remove-process-property indicator => removedp`

**Arguments**

- `indicator` A Lisp object.
- `process` A process.

**Values**

- `removedp` A generalized boolean.
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.

The following example schedules a timer to expire 15 minutes after the start of the program and every 5 minutes thereafter.

```
(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 248

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.

```scheme
(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.
The following example schedules a timer to expire 5 seconds after the call to `schedule-timer-relative` and every 5 seconds thereafter.

```lisp
(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.

```lisp
(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`
The function \texttt{semaphore-acquire} acquires \texttt{count} units from the semaphore \texttt{sem}.

It attempts to atomically decrement the semaphore's unit count by \texttt{count}. If this gives a non-negative result, then it changes the semaphore's unit count accordingly and returns true. The default value of \texttt{count} is 1.

However, if decrementing the semaphore's unit count would result in a negative number then \texttt{semaphore-acquire} waits until the semaphore's unit count is larger than \texttt{count} and tries again. If \texttt{wait-reason} is true, then it is used as the thread's \texttt{wait-reason} when waiting for the semaphore.
If `timeout` is `nil`, `semaphore-acquire` can wait forever. If `timeout` is true, it should be an integer. 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
`make-semaphore`
`semaphore-count`
`semaphore-release`
`semaphore-wait-count`

---

**semaphore-count**

*Summary*
Gets the current unit count of a semaphore.

*Package*
`mp`

*Signature*
`semaphore-count sem => count`

*Arguments*
`sem` A semaphore.

*Values*
`count` A non negative fixnum.

*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*
`make-semaphore`
`semaphore-acquire`
semaphore-name

Function

Summary  Gets the name of a semaphore.

Package  mp

Signature  semaphore-name sem => name

Arguments  sem  A semaphore.

Values  name  An object.

Description  The function semaphore-name returns the name that semaphore sem was given when it was created.

See also  make-semaphore

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 \texttt{count}
(which defaults to 1).

The returned \texttt{flag} is true if any other thread was waiting for the semaphore and false otherwise.

\textbf{See also} \texttt{make-semaphore}\newline \texttt{semaphore-acquire}\newline \texttt{semaphore-count}\newline \texttt{semaphore-wait-count}

\textbf{semaphore-wait-count} \hspace{1cm} \textit{Function}

\textbf{Summary} \hspace{1cm} Get the current wait count of a semaphore.

\textbf{Package} \hspace{1cm} \texttt{mp}

\textbf{Signature} \hspace{1cm} \texttt{semaphore-wait-count \textit{sem} => \textit{wait-count}}

\textbf{Arguments} \hspace{1cm} \texttt{\textit{sem} \hspace{1cm} A semaphore.}

\textbf{Values} \hspace{1cm} \texttt{\textit{wait-count} \hspace{1cm} A non negative fixnum.}

\textbf{Description} \hspace{1cm} The function \texttt{semaphore-wait-count} returns the current number of units that other threads are waiting for from the semaphore \textit{sem}. The value \textit{wait-count} is 0 if the semaphore has no thread waiting for it.

\textbf{Notes} \hspace{1cm} The value can change in the semaphore after calling \texttt{semaphore-wait-count}.

\textbf{See also} \texttt{make-semaphore}\newline \texttt{semaphore-acquire}\newline \texttt{semaphore-count}\newline \texttt{semaphore-release}
The MP Package

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 2, which is adequate if funcall-async and/or funcall-async-list are rarely used. 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.
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 the predicate or the wait, like lock-and-condition-variable-wait when return-function is not supplied.

simple-lock-and-condition-variable-wait does not take wait reason arguments, so you should give names to the lock 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

**symeval-in-process**

*Function*

**Summary**
Reads the value of symbol which is dynamically bound in a given process.

**Package**
mp

**Signature**
symeval-in-process symbol process => value, flag
(setf symeval-in-process) value symbol process => value
The function `symeval-in-process` reads the value of the symbol `symbol` in the process `process` if it is bound dynamically. The global value of `symbol` is never returned.

If `symbol` is not bound in `process`, then `value` and `flag` are both `nil`. If `symbol` is bound in `process` but `makunbound` has been called within the dynamic scope of the binding, `value` is `nil` and `flag` is `:unbound`. Otherwise, `value` is the value of `symbol` and `flag` is `t`.

In addition, the form

```lisp
(setf (symeval-in-process symbol process) value)
```

sets the value of `symbol` to `value` in `process`. It is an error if `process` has no binding for `symbol`. This `setf` form returns `value` as specified by Common Lisp.

The function `timer-expired-p` reads the value of the symbol `symbol` in the process `process` if it is bound dynamically. The global value of `symbol` is never returned.

If `symbol` is not bound in `process`, then `value` and `flag` are both `nil`. If `symbol` is bound in `process` but `makunbound` has been called within the dynamic scope of the binding, `value` is `nil` and `flag` is `:unbound`. Otherwise, `value` is the value of `symbol` and `flag` is `t`.

In addition, the form

```lisp
(setf (symeval-in-process symbol process) value)
```

sets the value of `symbol` to `value` in `process`. It is an error if `process` has no binding for `symbol`. This `setf` form returns `value` as specified by Common Lisp.

**Notes**

`symeval-in-process` is mostly intended for debugging. It is OK to call it on a thread known to be idle, or in `process-wait` or `process-stop`, but it should not be called while the thread is running.

**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
```
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

```lisp
(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`
- `schedule-timer-relative`
- `timer-name`
- `unschedule-timer`

**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*))

#<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
unnotice-fd

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

Summary: Unschedules a scheduled timer

Package: mp

Signature: unschedule-timer timer => result

Arguments: timer A timer

Values: result A timer or nil
The MP Package

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

```
(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**

`wait-processing-events timeout &key wait-reason wait-function wait-args => result`
Arguments

- **timeout**: A number or nil.
- **wait-reason**: A string.
- **wait-function**: A function designator.
- **wait-args**: A list.

Values

- **result**: t or nil

Description

The function **wait-processing-events** does not return until one of two conditions is met:

- **timeout** is non-nil and **timeout** seconds have passed.
  
  In this case, **result** is nil.

- **wait-function** returns a true value.
  
  In this case, **result** is t.

**wait-reason** provides the value returned by **process-whostate** when called on the current process.

**wait-function** is called periodically with arguments **wait-args**.

**wait-function** may be called many times and in several places. Therefore **wait-function** should be fast and make no assumptions about its dynamic context.

**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 **process-send**. In the latter case, the objects must be lists of the form (function . arguments), which cause function to be applied to 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 the lock is locked, as seen in the Process Browser.
- **timeout**
  A timeout period, in seconds.
- **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 the lock must be "sharing", that is, created with the argument **sharing** true in **make-lock**. It waits until **sharing-lock** is completely free, that is, the lock is not locked in a sharing mode and is not locked in exclusive mode by another thread. It then locks the lock **sharing-lock** in exclusive mode, evaluates **body** and unlocks the lock.

**Notes**
It is not possible to use 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 use exclusive-lock inside exclusive-lock of the same lock is determined by the **recursivep** argument in **make-lock**.

**See also**
- make-lock
- with-lock
with-interrupts-blocked

Macro

Summary
Evaluates code with interrupts blocked.

Package
mp

Signature
with-interrupts-blocked &body body => results

Arguments
body Code

Values
results Values returned by evaluating body.

Description
Evaluates body with interrupts blocked. This actually expands to

(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
allowing-block-interrupts

with-lock

Macro

Summary
Executes a body of code while holding a lock.

Package
mp

Signature
with-lock (lock &optional whostate timeout) &body body => result

Arguments
lock The lock.

whostate The status of the process while the lock is locked, as seen in the Process Browser.
timeout
A timeout period, in seconds.

body
The forms to execute.

Values
result
The result of executing body.

Description
with-lock executes body while holding the lock, and unlocks the lock when body exits. This is the recommended way of using locks. The value of body is returned normally. body is not executed if the lock could not be claimed, in which case, with-lock returns nil.

See also
make-lock
process-lock
process-unlock
with-exclusive-lock
with-sharing-lock

with-sharing-lock

Macro

Summary
Holds a lock in shared mode while executing a body of code.

Package
mp

Signature
with-sharing-lock (sharing-lock &optional whostate timeout)
&body body => results

Arguments
sharing-lock
A sharing lock.

whostate
The status of the process while the lock is locked, as seen in the Process Browser.

timeout
A timeout period, in seconds.

body
The forms to execute

Values
results
The values returned from evaluating body.
Description

The macro `with-sharing-lock` is like `with-lock`, but the lock must be "sharing" and the lock will be locked in shared mode. That means that other threads can also lock it in shared mode.

Before locking this waits for the 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`

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 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.

```lisp
(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  `without-preemption` is not supported in SMP LispWorks, that is on Microsoft Windows, Mac OS X, Linux, FreeBSD and x86/x64 Solaris platforms.
<table>
<thead>
<tr>
<th><strong>yield</strong></th>
<th><strong>Function</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
<td>Allows preemption to happen in low safety code.</td>
</tr>
<tr>
<td><strong>Package</strong></td>
<td>mp</td>
</tr>
<tr>
<td><strong>Signature</strong></td>
<td>yield</td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>None.</td>
</tr>
<tr>
<td><strong>Values</strong></td>
<td>None.</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Normally code compiled at safety 0 cannot be preempted because the necessary checks are omitted. This can be overcome by calling yield at regular intervals. Usually there is no need to call this if you use functions from the 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, yield can be useful. Note that process-allow-scheduling also allows preemption, but also checks the wait functions of other processes.</td>
</tr>
<tr>
<td><strong>See also</strong></td>
<td>process-allow-scheduling</td>
</tr>
</tbody>
</table>
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 {rule}*: = parsing-function
rule : = normal-rule | error-rule
normal-rule : = ( (non-terminal {grammar-symbol}*) {form}* )
error-rule : = ( (non-terminal :error) {form}* )
```

**Arguments**  
name The name of the parser.
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:

```lisp
(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.

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

```
(reuse "serial-port")
```

**Note:** this chapter applies only to LispWorks for Windows, and not the UNIX, Linux or Mac OS X platforms.

**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 function `open-serial-port` attempts to open the serial port `name` and return a `serial-port` object.

`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`.

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 or 2. The value of `parity` should be one of the keywords `:even`, `:mark`, `:none`, `:odd` 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. 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.

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:

\[ \text{basetimeout} + \text{nchars} \times \text{byte_timeout} \]

The default value of each of read-total-base-timeout, read-total-byte-timeout, write-total-base-timeout and write-total-byte-timeout is 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

close-serial-port

set-serial-port-state

close-serial-port

Function

Summary
Closes a serial port

Package serial-port

Signature close-serial-port serial-port

Arguments serial-port A serial-port object.

Description The function close-serial-port closes the serial port associated with the given serial-port object.

If serial-port is already closed, an error is signaled.

See also open-serial-port
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.

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

Signature  serial-port-input-available-p serial-port => result

Arguments  serial-port  A serial-port object.

Values  result  A boolean.
This chapter applies only to LispWorks for Windows

**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**
`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**

**Function**

**Summary**
Waits for some aspect of the state of a serial port to change.

**Package**
serial-port
The SERIAL-PORT Package

This chapter applies only to LispWorks for Windows

### wait-serial-port-state

**Signature**

\[
\text{wait-serial-port-state} \quad \text{serial-port} \quad \text{keys} \quad \&\text{key} \quad \text{timeout} \quad \Rightarrow \quad \text{result}
\]

**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`, `:r1sd` 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

**Function**

**Summary**

Writes a character to a serial port.

**Package**

`serial-port`

**Signature**

\[
\text{write-serial-port-char} \quad \text{char} \quad \text{serial-port} \quad \&\text{optional} \quad \text{timeout-error-p} \quad \Rightarrow \quad \text{char}
\]

**Arguments**

- **char**
  - A character.
- **serial-port**
  - A `serial-port` object.
- **timeout-error-p**
  - A boolean.

**Values**

- **char**
  - A character.
This chapter applies only to LispWorks for Windows

Description

The function **write-serial-port-char** writes the character \texttt{char} to the serial port associated with \texttt{serial-port}, and returns \texttt{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 \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}.

See also

**write-serial-port-string**

Function

Summary

Writes a string to a serial port.

Package

\texttt{serial-port}

Signature

\texttt{write-serial-port-string string serial-port &optional timeout-error-p &key start end => nwritten}

Arguments

\texttt{string} A string.

\texttt{serial-port} A \texttt{serial-port} object.

\texttt{timeout-error-p} A boolean.

\texttt{start, end} Bounding index designators for \texttt{string}.

Values

\texttt{result} The string \texttt{string} or \texttt{nil}.

Description

The function **write-serial-port-string** writes characters from the subsequence of \texttt{string} bounded by \texttt{start} and \texttt{end} to the serial port associated with \texttt{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 `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 `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-char`
The SQL Package

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 and Mac OS X, Common SQL is included only in LispWorks Enterprise Edition.

**add-sql-stream**

*Function*

**Summary**  Adds a stream to the broadcast list for SQL commands or results traffic.

**Package**  sql

**Signature**  add-sql-stream stream &key type database => added-stream

**Arguments**  

- *stream*  A stream, or t.
- *type*  A keyword.
The SQL Package

This chapter applies to the Enterprise Edition only

47

47 The SQL Package

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.
This chapter applies to the Enterprise Edition only

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

(loop for tab in (sql:list-tables)
  do
    (loop for att in (sql:list-attributes tab)
      do
        (format t "Table ~S Attribute ~S Type ~S"
          tab att
          (sql:attribute-type att tab)))))

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*`
This chapter applies to the Enterprise Edition only

**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.

Example This example changes records in a database, and uses commit to make those changes permanent.
(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**
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 => 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.
This chapter applies to the Enterprise Edition only

`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`.

**Values**

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>database</code></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 287 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.

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 288 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 293 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 equalp. 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:
This chapter applies to the Enterprise Edition only

: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 \texttt{\textasteriskcentered connect-if-exists}.\textasteriskcentered

\texttt{interface} is used if connect needs to display a dialog to ask the user for username and password. If \texttt{interface} is a CAPI element, this is used. If \texttt{interface} is any other value (the default value is \texttt{nil}), and connect is called in a process which is associated with a CAPI interface, then this CAPI interface is used. \texttt{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.

\texttt{encoding} specifies the encoding to use in the connection. The value should be a keyword naming an acceptable encoding, or \texttt{nil} (the default). The value :unicode is accepted for all database-types, and this will try to make a connection that can support sending and retrieving double-byte string values. Other values are database-type specific:

:mysql

If \texttt{encoding} is \texttt{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 \texttt{encoding} are :unicode, :utf-8, :ascii, :latin-1, :euc and :sjis.
:unide 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>:unicode</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>EUC_JP</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.

: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.

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-transactions, "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.

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:

(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")
```

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.
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.
This chapter applies to the Enterprise Edition only

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.
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

(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.

**Values**
None.

**Description**
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`. 
This chapter applies to the Enterprise Edition only

*type*, if non-nil, is used as argument to TYPE in the SQL statement:

```sql
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:

```sql
(create-table [manager]'
  `(([id] (char 10) not-null)
   ([salary] (number 8 2))))
```

is equivalent to the following SQL:

```sql
CREATE TABLE MANAGER
(ID CHAR(10) NOT NULL,SALARY NUMBER(8,2))
```

**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.

```lisp
```

See also `create-index`
`create-table`
`*default-database*`
`drop-view`

create-view-from-class Function

Summary Creates a view in a database based on a class that defines the view.

Package `sql`
This chapter applies to the Enterprise Edition only

**Signature**

```
create-view-from-class class &key database =>
```

**Arguments**

- `class`: A class.
- `database`: A database.

**Values**

None.

**Description**

The function `create-view-from-class` creates a view in `database` based on `class` which defines the view. The argument `database` has a default value of `*default-database*`.

**See also**

- `*default-database*`
- `drop-view-from-class`
- `create-view`

---

**database-name**

*Function*

**Summary**

Returns the name of a database.

**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`
The SQL Package

This chapter applies to the Enterprise Edition only

**Functions**

**decode-to-db-standard-date**
**decode-to-db-standard-timestamp**

**Summary**  Convert Lisp universal time to standard SQL DATE and TIMESTAMP.

**Package**  sql

**Signature**

\[
\begin{align*}
\text{decode-to-db-standard-date} & \quad \text{universal-time} \ & \text{key} \ & \text{stream} \ & \text{quoted} \Rightarrow \text{date} \\
\text{decode-to-db-standard-timestamp} & \quad \text{universal-time} \ & \text{key} \ & \text{stream} \ & \text{quoted} \Rightarrow \text{timestamp}
\end{align*}
\]

**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`. 
This chapter applies to the Enterprise Edition only

See also
- encode-db-standard-date
- encode-db-standard-timestamp
- connect
- “Working with date fields” on page 317

**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
defode-to-db-standard-date
defode-to-db-standard-timestam
connect
“Working with date fields” on page 317

*default-database*

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*

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 287.
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

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.
This chapter applies to the Enterprise Edition only

**: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.

**: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 set is `nil` the slot will contain a single instance.

The default value of set is `t`.

The syntax for `:home-key` and `:foreign-key` means that an object from a join class will only be 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. See the second example below.

The `: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 `:type` is mandatory for slots with `:db-kind :base` or `:key.` Some methods are provided for type checking and type conversion. For example, a `:type` specifier of `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 type, and correspond to the SQL type shown:

```
(string n) CHAR(n)
integer INTEGER
(integer n) INTEGER(n)
```
This chapter applies to the Enterprise Edition only

<table>
<thead>
<tr>
<th>float</th>
<th>FLOAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(float n)</td>
<td>FLOAT(n)</td>
</tr>
</tbody>
</table>

`sql:universal-time`  
TIMESTAMP

**Class Options**

`def-view-class` recognizes the following class options in addition to the standard class options defined for `defclass`:

`(:base-table table-name)`  
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.
the SQL Package

This chapter applies to the Enterprise Edition only

(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 name)
     :foreign-key
     (schema-name class-name slot-name)
     :join-class flex-property
     :retrieval :deferred))
  (:base-table flex_slot))
(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**

\[
\text{delete-instance-records} \ instances \Rightarrow
\]

**Arguments**

\[
\text{instance} \quad \text{An instance of a View Class.}
\]

**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**

\[
\text{delete-records} \ &key \ from \ where \ database \Rightarrow
\]

**Arguments**

\[
\text{from} \quad \text{A database table.}
\]

\[
\text{where} \quad \text{A SQL conditional statement.}
\]

\[
\text{database} \quad \text{A database.}
\]

**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*. 

1391
The SQL Package

This chapter applies to the Enterprise Edition only

See also

*default-database*
insert-records
update-records

delete-sql-stream

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
*default-database*
list-sql-streams
sql-recording-p
sql-stream
This chapter applies to the Enterprise Edition only

\begin{verbatim}
start-sql-recording
stop-sql-recording
\end{verbatim}

\textbf{destroy-prepared-statement} \textit{Function}

\textbf{Summary} Destroys a prepared-statement and frees its resources.

\textbf{Package} \textit{sql}

\textbf{Signature} \texttt{destroy-prepared-statement} \texttt{prepared-statement} \texttt{=> nil}

\textbf{Arguments} \texttt{prepared-statement}

A prepared-statement.

\textbf{Description} The function \texttt{destroy-prepared-statement} destroys the prepared-statement \texttt{prepared-statement} and frees its resources. It should be called before closing the database associated with the prepared-statement. A destroyed prepared-statement can be reused by calling \texttt{set-prepared-statement-variables} with a new database.

\texttt{destroy-prepared-statement} always returns \texttt{nil}.

\textbf{See also} \texttt{prepare-statement} \texttt{set-prepared-statement-variables}

\textbf{disable-sql-reader-syntax} \textit{Function}

\textbf{Summary} Turns off square bracket syntax.

\textbf{Package} \textit{sql}

\textbf{Signature} \texttt{disable-sql-reader-syntax} \texttt{=>}

\textbf{Arguments} None.
The function disable-sql-reader-syntax turns off square bracket syntax and sets state so that restore-sql-reader-syntax-state will make the syntax disabled if it is consequently enabled.

See also enable-sql-reader-syntax locally-disable-sql-reader-syntax locally-enable-sql-reader-syntax restore-sql-reader-syntax-state

**disconnect**

**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 323 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*}. 
See also  
create-index  
drop-table

**drop-table**

**Function**

**Summary**  
Deletes a table from a database.

**Package**  
sql

**Signature**  
drop-table table &key database =>

**Arguments**  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>The name of a table.</td>
</tr>
<tr>
<td>database</td>
<td>A database.</td>
</tr>
</tbody>
</table>

**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**  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>view</td>
<td>A view.</td>
</tr>
<tr>
<td>database</td>
<td>A database.</td>
</tr>
</tbody>
</table>

**Values**  
None.
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*
enable-sql-reader-syntax  

**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

**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.
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
(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.
- `errorp` A boolean. Default value: `t`.

**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 \textit{count} is the number of matches. If no matching database is found and \textit{errorp} is \texttt{nil}, then \texttt{nil} is returned. If none, or more than one, matching databases are found and \textit{errorp} is true, then an error is signaled.

If the argument \textit{database} is a database, it is simply returned.

\textbf{See also} \texttt{connect} \newline \texttt{connected-databases} \newline \texttt{database-name} \newline \texttt{disconnect} \newline \texttt{status}

\textbf{initialize-database-type} \hspace{1cm} \textit{Function}

\textbf{Summary} \hspace{1cm} Initializes a database type.

\textbf{Package} \hspace{1cm} \texttt{sql}

\textbf{Signature} \hspace{1cm} \texttt{initialize-database-type \&key database-type => type}

\textbf{Arguments} \hspace{1cm} \texttt{database-type} A database type.

\textbf{Values} \hspace{1cm} \texttt{type} A database type.

\textbf{Description} \hspace{1cm} The function \texttt{initialize-database-type} initializes a database type by loading code and appropriate database libraries according to the value of \textit{database-type}. If \texttt{*default-database-type*} is not initialized, this function initializes it. It adds \textit{database-type} to the list of initialized types. The initialized database type is returned.

\textbf{Example} \hspace{1cm} The following example shows how to use \texttt{initialize-database-type} to initialize the \texttt{: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:

\[
\text{INSERT INTO PERSON}
\text{VALUES ("abc", "Joe", "Bloggs", 10000, 3000, NULL, "plumber")}
\]

In the second example, the LispWorks expression

(insert-records :into [person]
   :attributes (person_id income surname occupation)
   :values ('("aaa" 10 "jim" "plumb"))

1403
is equivalent to the following SQL:

```sql
INSERT INTO PERSON
   (PERSON_ID, INCOME, SURNAME, OCCUPATION)
VALUES ('aaa', 10, 'jim', 'plumb')
```

The following example demonstrates how to use `:av-pairs`.

```lisp
(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

```lisp
(select ... :refresh t)
```

then add an `instance-refresh` method specializing on your subclass of `standard-db-object`. 
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 informa-
tion 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

\[
\text{(loop for tab in (sql:list-tables) do}
\text{(loop for type-info in (sql:list-attribute-types tab) do}
\text{(format t "-&Table ~S Attribute ~S Type ~S"
\text{tab (first type-info) (second type-info))}})
\]

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*.
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**

*Function*

**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

**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

**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

buffered-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 =>
**locally-disable-sql-reader-syntax**

<table>
<thead>
<tr>
<th>Arguments</th>
<th>None.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>None.</td>
</tr>
<tr>
<td>Description</td>
<td>The function <code>locally-disable-sql-reader-syntax</code> turns off square bracket syntax and does not change syntax state. This ensures that <code>restore-sql-reader-syntax-state</code> restores the current enable/disable state.</td>
</tr>
<tr>
<td>Example</td>
<td>The intended use of <code>locally-disable-sql-reader-syntax</code> is in a file:</td>
</tr>
<tr>
<td></td>
<td><code>. (locally-disable-sql-reader-syntax)</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Lisp code not using [...] syntax&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>. (restore-sql-reader-syntax-state)</code></td>
</tr>
<tr>
<td>See also</td>
<td><code>disable-sql-reader-syntax</code></td>
</tr>
<tr>
<td></td>
<td><code>enable-sql-reader-syntax</code></td>
</tr>
<tr>
<td></td>
<td><code>locally-enable-sql-reader-syntax</code></td>
</tr>
<tr>
<td></td>
<td><code>restore-sql-reader-syntax-state</code></td>
</tr>
</tbody>
</table>

**Function**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Turns on square bracket syntax and does not change syntax state.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td><code>sql</code></td>
</tr>
<tr>
<td>Signature</td>
<td><code>locally-enable-sql-reader-syntax</code></td>
</tr>
</tbody>
</table>
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:

```lisp
#.(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**

```lisp
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 \texttt{var}. The record is represented in Lisp as a list. Destructuring can be used in \texttt{var} to bind variables to specific attributes of the records resulting from \texttt{query-expression}. In conjunction with the panoply of existing clauses available from the \texttt{loop} macro, the new iteration clause provides an integrated report generation facility.

The additional loop keywords \texttt{not-inside-transaction} and \texttt{get-all} may be useful when fetching many records through a connection with \texttt{database-type: mysql}. See the section “Special considerations for iteration functions and macros” on page 323 for details.

Example
This extended \texttt{loop} example performs the following on each record returned as a result of a query: bind \texttt{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.

\begin{verbatim}
(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)))
\end{verbatim}
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  

map-query output-type-spec function query-exp &key database not-inside-transaction get-all => result

Arguments

output-type-spec  The output type specification.
result-type  The result sequence type.
function  A function.
query-exp  A SQL query or a prepared-statement containing a query.
database  A database.
not-inside-transaction  A generalized boolean.
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 argu-
This chapter applies to the Enterprise Edition only

...ment 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 323 for details.

**Example**

This example binds `name` to each name in the employee table and prints it.

```
(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 path-
name 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

Package sql

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
This chapter applies to the Enterprise Edition only

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 327 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 327 for more information.

**ora-lob-char-set-form**

*Function*

**Summary**
Returns the character set form of a LOB.
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.

ccharset 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 327 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

\[ \text{db-charset-id} \]

A non-negative number.

Description

The function \texttt{ora-lob-char-set-id} returns the database character set identifier of the LOB underlying \textit{lob-locator}.

\[ \text{db-charset-id} \] is 0 for a binary LOB.

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}.

Notes

1. This is a direct call to OCILobCharSetID.

2. \texttt{ora-lob-char-set-id} is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 327 for more information.

\texttt{ora-lob-close} Function

Summary

Closes an opened LOB.

Package

\texttt{sql}

Signature

\texttt{ora-lob-close lob-locator &key errorp}

Arguments

\textit{lob-locator} A LOB locator.

\textit{errorp} A generalized boolean.

Description

The function \texttt{ora-lob-close} closes a LOB which has been opened by \texttt{ora-lob-open}.

For more information see \texttt{ora-lob-open}.

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}.

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 327 for more information.

See also `ora-lob-open`

### Functions

#### `ora-lob-copy`

**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
This chapter applies to the Enterprise Edition only

end and 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.

ora-lob-append 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 OCILOBCopy.

2. This function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 327 for more information.

See also ora-lob-load-from-file

ora-lob-create-empty Function

Summary Creates an empty LOB.

Package sql

Signature ora-lob-create-empty &key db type => lob-locator

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>A database.</td>
</tr>
<tr>
<td>type</td>
<td>A Lisp object.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lob-locator</td>
<td>A LOB locator.</td>
</tr>
</tbody>
</table>

Description The function ora-lob-create-empty creates an empty LOB object and returns a LOB locator for it.
If \textit{type} is \texttt{:lob} then \texttt{ora-lob-create-empty} creates a LOB of type BLOB/CLOB. If \textit{type} is any other value, it creates a file LOB. The default value of \textit{type} is \texttt{:lob}.

Empty LOBs can be put in the database by passing them to \texttt{insert-records} or \texttt{update-records}. However, the preferred approach is to use the Oracle SQL function \texttt{EMPTY_BLOB} as described in the section “Inserting empty LOBs” on page 328.

The default value of \textit{db} is the value of \texttt{*default-database*}.

Notes

\texttt{ora-lob-create-empty} is available only when the ”oracle” module is loaded. See the section “Oracle LOB interface” on page 327 for more information.

\textbf{ora-lob-create-temporary} \\

\textit{Function}

\textbf{Summary} Creates a temporary LOB.

\textbf{Package} sql

\textbf{Signature} \texttt{ora-lob-create-temporary db-or-lob-locator &key errorp cache session-duration clob-p => lob-locator}

\textbf{Arguments} \\
\texttt{db-or-lob-locator} \\
\hspace{1em} A database or a LOB locator.

\texttt{errorp} \\
\hspace{1em} A generalized boolean.

\texttt{cache} \\
\hspace{1em} A generalized boolean.

\texttt{session-duration} \\
\hspace{1em} A generalized boolean.

\texttt{clob-p} \\
\hspace{1em} A generalized boolean.

\textbf{Values} \\
\texttt{lob-locator} \\
\hspace{1em} 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_DURATI0N_SESSION`, otherwise it uses `OCI_DURATI0N_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 327 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`
The SQL Package

This chapter applies to the Enterprise Edition only

Arguments

- lob-locator: A LOB locator.
- errorp: A generalized boolean.

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 327 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.
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 327 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 327 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 poci-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 327 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 327 for more information.

ora-lob-file-close  
Function

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 327 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 327 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 errrop => result

Arguments
lob-locator A LOB locator.
errrop A generalized boolean.

Values
result A boolean.

Description
The function ora-lob-file-exists returns 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 errrop is true, an error is signaled. If errrop is false, the function returns an object of type sql-database-error. The default value of errrop is 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 327 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 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 OCILobFileGetName.
2. ora-lob-file-get-name is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 327 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 327 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 327 for more information.

See also ora-lob-file-close

ora-lob-file-set-name

Function

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.
This chapter applies to the Enterprise Edition 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 to OCILobFileSetAlias.
2. \texttt{ora-lob-file-set-name} is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 327 for more information.

\textbf{ora-lob-flush-buffer}

\textit{Function}

\textbf{Summary}
Flushes the buffer of the Oracle client.

\textbf{Package}
\texttt{sql}

\textbf{Signature}
\texttt{ora-lob-flush-buffer} \texttt{lob-locator} \texttt{&key} \texttt{free-buffer} \texttt{errorp}

\textbf{Arguments}
\begin{itemize}
\item \texttt{lob-locator} A LOB locator.
\item \texttt{free-buffer} A generalized boolean.
\item \texttt{errorp} A generalized boolean.
\end{itemize}

\textbf{Description}
The function \texttt{ora-lob-flush-buffer} flushes the buffer that is used by the Oracle client.

If \texttt{free-buffer} is true, it also frees the buffer. The default value of \texttt{free-buffer} is \texttt{nil}.

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 OCILobFlushBuffer.
2. \texttt{ora-lob-flush-buffer} is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 327 for more information.
See also  
ora-lob-enable-buffering

**ora-lob-free**  
*Function*

**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 327 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 327 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 to `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.
Notes  

`ora-lob-get-buffer` is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 327 for more information.

Example  

This first example illustrates reading using the buffer obtained by `ora-lob-get-buffer`. You have a foreign function:

```lisp
my_chunk_processor(char *data, int size)
```

with this FLI definition:

```lisp
(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.

```lisp
(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.

```lisp
(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 327 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 327 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.

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.
Notes  
ora-lob-internal-lob-p is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 327 for more information.

### 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 327 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

\[ \text{errorp} \quad \text{A generalized boolean.} \]

Values
\[ \text{result} \quad \text{A boolean.} \]

Description
The function \texttt{ora-lob-is-open} returns \texttt{t} if the LOB pointed to by \texttt{lob-locator} is opened (by \texttt{ora-lob-open}).

\texttt{ora-lob-is-open} 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 to OCILOBIsOpen.
2. \texttt{ora-lob-is-open} is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 327 for more information.

See also \texttt{ora-lob-open}

\textbf{ora-lob-is-temporary} 

Function

Summary
The predicate for whether a LOB is temporary.

Package \texttt{sql}

Signature
\texttt{ora-lob-is-temporary lob-locator &key errorp => result}

Arguments
\begin{itemize}
  \item \texttt{lob-locator} \quad \text{A LOB locator.}
  \item \texttt{errorp} \quad \text{A generalized boolean.}
\end{itemize}

Values
\begin{itemize}
  \item \texttt{result} \quad \text{A boolean.}
\end{itemize}

Description
The function \texttt{ora-lob-is-temporary} returns \texttt{t} if the LOB underlying \texttt{lob-locator} is temporary, that is, it was created by \texttt{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. This is a direct call to `OCILobIsTemporary`.
2. `ora-lob-is-temporary` is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 327 for more information.

**See also**

`ora-lob-create-temporary`

### Function

**ora-lob-load-from-file**

**Summary**

Loads data from a file LOB into a LOB.

**Package**

`sql`

**Signature**

`ora-lob-load-from-file dest-lob-locator src-lob-file amount &key src-offset dest-offset errorp`

**Arguments**

- `dest-lob-locator` An internal LOB locator.
- `src-lob-file` A file LOB locator.
- `amount` A non-negative integer.
- `src-offset` A non-negative integer.
- `dest-offset` A non-negative integer.
- `errorp` A generalized boolean.

**Description**

The function `ora-lob-load-from-file` loads the data from the `src-lob-file` into the destination LOB pointed to by `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 327 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 p-oci-lob-locator or p-oci-file.

Notes
ora-lob-locator is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 327 for more information.

ora-lob-locator-is-init

Summary
The predicate for whether a LOB is initialized.

Package
sql

Signature
ora-lob-locator-is-init lob-locator &key errorp => result

Arguments
lob-locator A LOB locator.
errorp A generalized boolean.

Values
result A boolean.

Description
The function ora-lob-locator-is-init returns t if the LOB locator lob-locator is initialized.

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 OCILobIsInit.
2. ora-lob-locator-is-init is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 327 for more information.

ora-lob-open

Summary
Opens a LOB.
This chapter applies to the Enterprise Edition only

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 327 for more information.

See also   ora-lob-close
           ora-lob-is-open

ora-lob-read-buffer
Function
Summary  Reads from a LOB into a buffer.
Package  sql

Signature  ora-lob-read-buffer  lob-locator  offset  amount  buffer  &key
            buffer-offset  csid  =>  amount-read,  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-read  A non-negative integer.
eof-or-error-p  A boolean or an error object.

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 com-
parison (unsigned-byte 8) and base-char are considered
as the same.

If the buffer buffer is not static, there is some additional over-
head. For small amounts of data, this is probably insignifi-
cant.
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 327 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
Function

ora-lob-read-into-plain-file

Summary
WritesthecontentsofaLOBintoafile.

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  \texttt{ora-lob-read-into-plain-file} is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 327 for more information.

See also  \texttt{ora-lob-write-from-plain-file}

\textbf{ora-lob-read-foreign-buffer} \hspace{1cm} \textit{Function}

\textbf{Summary}  Reads from a LOB into a foreign buffer.

\textbf{Package}  sql

\textbf{Signature}  \texttt{ora-lob-read-foreign-buffer lob-locator offset amount foreign-buffer buffer-length &key buffer-offset csid => amount-read, eof-or-error-p}

\textbf{Arguments}  
\begin{itemize}
  \item \texttt{lob-locator}  A LOB locator.
  \item \texttt{offset}  A non-negative integer or \texttt{nil}.
  \item \texttt{amount}  A non-negative integer.
  \item \texttt{foreign-buffer}  A FLI pointer.
  \item \texttt{buffer-length}  A non-negative integer.
  \item \texttt{buffer-offset}  A non-negative integer.
  \item \texttt{csid}  A Character Set ID
\end{itemize}

\textbf{Values}  
\begin{itemize}
  \item \texttt{amount-read}  A non-negative integer.
  \item \texttt{eof-or-error-p}  A boolean or an error object.
\end{itemize}

\textbf{Description}  The function \texttt{ora-lob-read-foreign-buffer} reads from the LOB pointed to by \texttt{lob-locator} into the foreign buffer \texttt{foreign-buffer}.

This is just like \texttt{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 327 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 327 for more information.

ora-lob-trim

Function

Summary
Trims an internal LOB.
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>ora-lob-trim lob-locator new-size &amp;key errorp</td>
</tr>
<tr>
<td>Arguments</td>
<td>lob-locator A LOB locator.</td>
</tr>
<tr>
<td></td>
<td>new-size A non-negative integer.</td>
</tr>
<tr>
<td></td>
<td>errorp A generalized boolean.</td>
</tr>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
<tr>
<td>Notes</td>
<td>1. This is a direct call to OCILobTrim.</td>
</tr>
<tr>
<td></td>
<td>2. ora-lob-trim is available only when the &quot;oracle&quot; module is loaded. See the section “Oracle LOB interface” on page 327 for more information.</td>
</tr>
</tbody>
</table>

**ora-lob-write-buffer**

Function

<table>
<thead>
<tr>
<th>Summary</th>
<th>Writes a buffer to a LOB.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>sql</td>
</tr>
<tr>
<td>Signature</td>
<td>ora-lob-write-buffer lob-locator offset amount buffer &amp;key buffer-offset csid =&gt; amount-written, eof-or-error-p</td>
</tr>
<tr>
<td>Arguments</td>
<td>lob-locator A LOB locator.</td>
</tr>
</tbody>
</table>
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.
$csid$ 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 329.
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 327 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 $nil$. 
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 327 for more information.

See also

`ora-lob-read-into-plain-file`

**Function**

`ora-lob-write-foreign-buffer`  
**Summary**

Writes a foreign buffer to a LOB.

**Package**

`sql`
This chapter applies to the Enterprise Edition only

Signature

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ora-lob-write-foreign-buffer</td>
<td>A foreign type representing objects in the Oracle interface.</td>
</tr>
<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>foreign-buffer</td>
<td>A FLI pointer.</td>
</tr>
<tr>
<td>buffer-length</td>
<td>A non-negative integer.</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>

Arguments

<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>foreign-buffer</td>
<td>A FLI pointer.</td>
</tr>
<tr>
<td>buffer-length</td>
<td>A non-negative integer.</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>

Values

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount-written</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-write-foreign-buffer** writes to the LOB pointed to by **lob-locator** from **buffer**.

This is just like **ora-lob-write-buffer** except that it writes the LOB locator from a foreign buffer.

**foreign-buffer** is a FLI pointer to a buffer, which must be of size at least **buffer-length**.

Notes

**ora-lob-write-foreign-buffer** is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 327 for more information.

See also

**ora-lob-get-buffer**

**ora-lob-write-buffer**
47 The SQL Package  This chapter applies to the Enterprise Edition only

Package sql
Description See “Interactions with foreign calls” on page 331 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 331 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 331 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 331 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 331 for details.

**prepare-statement**

*Function*

**Summary**  Returns a prepared-statement object for a sql-exp in a database.

**Package**  sql

**Signature**

```
prepare-statement sql-exp &key database variable-types count
flatp result-types => 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.

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 variable, 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:

```
(setq ps
  (sql:prepare-statement
   "insert into TABLETWO values(:1, :2)"))
```

Then insert records into TABLETWO (which has two columns) by repeatedly doing:
This chapter applies to the Enterprise Edition only

```
(sql:set-prepared-statement-variables
  ps
  (list value1 value2))

(sql:execute-command ps))
```

See also
query
do-query
simple-do-query
map-query
select
set-prepared-statement-variables
destroy-prepared-statement

---

**Function**

**print-query**

**Summary**
Prints a tabulated version of records resulting from a query.

**Package**
sql

**Signature**
print-query query-exp &key titles formats sizes stream database =>

**Arguments**
- `query-exp` A SQL query expression.
- `titles` A list of strings.
- `formats` A list of strings.
- `sizes` A list.
- `stream` An output stream.
- `database` A database.

**Values**
None.

**Description**
The function **print-query** takes a symbolic SQL query expression and formatting information and prints onto `stream` a table containing the results of the query.
A list of strings to use as column headings is given by \textit{titles}, which has a default value of \texttt{nil}.

The \textit{formats} argument is a list of format strings used to print each attribute, and has a default value of \texttt{t}, which means that ~\texttt{A} or ~\texttt{VA} are used if sizes are provided or computed.

The field sizes are given by \textit{sizes}. It has a default value of \texttt{t}, which specifies that minimum sizes are computed.

The output stream is given by \textit{stream}, which has a default value of \texttt{t}. This specifies that *\texttt{standard-output}* is used.

\textbf{Examples}
The following call prints out two even columns of names and salaries:

\begin{verbatim}
(print-query [select [surname] [income] :from [person]]
  :titles '("NAME" "SALARY")
\end{verbatim}

\textbf{See also}
\begin{itemize}
\item map-query
\item print-query
\item select
\end{itemize}

\section*{query}

\begin{description}
\item[Summary] Queries a database and returns a list of values.
\item[Package] sql
\item[Signature] \texttt{query \textit{sql-exp} \&key \textit{database} \textit{result-types} \textit{flatp} \Rightarrow \textit{result-list}, \textit{field-names}}
\item[Arguments] \begin{itemize}
\item \texttt{sql-exp} A SQL query statement or a prepared-statement containing a query.
\item \texttt{database} A database.
\item \texttt{result-types} A list of symbols.
\item \texttt{flatp} A boolean.
\end{itemize}
\end{description}
This chapter applies to the Enterprise Edition only

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
<th>Example</th>
<th>See also</th>
</tr>
</thead>
<tbody>
<tr>
<td>result-list</td>
<td>A list of values.</td>
<td>(sql:query &quot;select * from some_table&quot;</td>
<td>do-query</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:result-types '(nil :string))</td>
<td>execute-command</td>
</tr>
<tr>
<td>field-names</td>
<td>A list of strings.</td>
<td>(sql:query [select [*]</td>
<td>lob-stream</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:from [some_table]</td>
<td>loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:result-types '(nil :string)])</td>
<td>map-query</td>
</tr>
<tr>
<td></td>
<td>The function query is the basic SQL query function. It queries the database specified by database with a SQL query statement given by sql-exp.</td>
<td>The following two queries, on a table whose second column contains dates that we want to return as strings, are equivalent:</td>
<td>prepare-statement</td>
</tr>
<tr>
<td></td>
<td>The argument database defaults to <em>default-database</em>.</td>
<td>(sql:query &quot;select * from some_table&quot;</td>
<td>select</td>
</tr>
<tr>
<td></td>
<td>result-types is a list of symbols such as :string and :integer, one for each field in the query, which are used to specify the types to return.</td>
<td>:result-types '(nil :string))</td>
<td>simple-do-query</td>
</tr>
<tr>
<td>flatp</td>
<td>is used as in select.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>result-list is a list of values as per select, and field-names is a list of field names selected in sql-exp.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


**Function**

**reconnect**

**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 `connect`). If no such database is found, then if `error` and `database` are both non-nil an error is signaled, otherwise `reconnect` returns `nil`.

`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 `force` is `nil`, an error is signaled if the database connection has been lost.

**Notes**
`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).
This chapter applies to the Enterprise Edition only

See also

connect
connected-databases
*default-database*

**restore-sql-reader-syntax-state**  
*Function*

Summary  
Sets the enable/disable square bracket syntax state to reflect the last call to either `disable-sql-reader-syntax` or `enable-sql-reader-syntax`.

Package  
sql

Signature  
`restore-sql-reader-syntax-state`

Arguments  
None.

Values  
None.

Description  
The function `restore-sql-reader-syntax-state` sets the enable/disable state of the square bracket syntax to reflect the last call to either `enable-sql-reader-syntax` or `disable-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`

Signature: `select &rest selections &key all set-operation distinct from result-types flatp where group-by having database order-by refresh for-update => result-list`

Arguments:
- `selections`: A set of database identifiers or strings or a prepared-statement.
- `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 t, :nowait, a string or a list.

Values result-list  A list of selections.

Description
The function `select` selects data from `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 `selections`. By default, the objects will each be represented as lists of attribute values.

The argument `selections` consists either of database identifiers, type-modified database identifiers or literal strings.

A type-modified database identifier is an expression such as `[foo :string]` which means that the values in column `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 321. 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.
The $flatp$ argument, 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 329.

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`.

Examples

The following is a potential query and result:

```lisp
(defun select-and-print-persons ()
  (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:

```lisp
(defun select-persons-flatp ()
  (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:
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 1KB of data from the first LOB returned by a query on an Oracle table containing a column of type LOB:

(let* ((array
    (make-array 1024
      :element-type '(unsigned-byte 8)))
   (lobs (sql:select [my-lob-column :input-stream]
       :from [mytable] :flatp t)))
 (read-sequence array (car lobs)))

See also
instance-refreshed
lob-stream
prepare-statement
print-query

set-prepared-statement-variables

Function

Summary
Sets the values of the bind variables in a prepared-statement.
This chapter applies to the Enterprise Edition only

Package sql

Signature

set-prepared-statement-variables prepared-statement &key
database values => prepared-statement

Arguments

preparedStatement

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 prepared-statement to the objects in the list given by values. The length of values must equal the number of bind-variables in the prepared-statement (that is, the supplied or computed count in prepare-statement). If database is supplied, then the 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

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.
The SQL Package

This chapter applies to the Enterprise Edition only

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
               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 323 for details.
Example

```lisp
(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
query
select

---

**sql**

*Function*

**Summary**
Generates SQL from a set of expressions.

**Package**
sql

**Signature**

```lisp
sql &rest args => sql-expression
```

**Arguments**
arg
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:

```lisp
string => (format nil "'-A'" x)
```
null => "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

sql-connection-error

Condition

Package sql
Superclasses sql-database-error
Subclasses
sql-fatal-error
sql-timeout-error
Description
The condition class sql-connection-error is used to signal an error with the connection to the database.

sql-database-data-error

Condition

Package sql
Superclasses sql-database-error
Description
The condition class sql-database-data-error is used to signal an error with the data given. This means either a syntax error or things like accessing a non-existent table.
This chapter applies to the Enterprise Edition only

It signifies an error that must be fixed for the code to work.

**sql-database-error**  
*Condition*

*Package*  
`sql`

*Superclasses*  
`simple-error`

*Subclasses*  
`sql-connection-error`  
`sql-database-data-error`  
`sql-temporary-error`

*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\-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.

Notes  
\*sql\-enlarge\-static\* is applicable in LispWorks for UNIX only (not LispWorks for Linux, FreeBSD, Mac OS X or x86/x64 Solaris).

\sql\\-expression  

**Function**

Summary  
Generates a SQL expression from the given keywords.

Package  
sql

Signature  
\sql\\-expression \&key string table alias attribute type  
\=> sql\-result

Arguments  
string A string.

\table A table in a database.

alias A table alias.

attribute An attribute.

type A type.

Values  
sql\-result A SQL expression.
This chapter applies to the Enterprise Edition only

**Description**  
The function `sql-expression` generates a SQL expression from the given keywords.

Valid combinations of the arguments are:

- `string`
- `table`
- `table` and `alias`
- `table` and `attribute`
- `table`, `attribute`, and `type`
- `table or alias`, and `attribute`
- `table or alias`, and `attribute` and `type`
- `attribute`
- `attribute` and `type`

**See also**  
`sql`  
`sql-operation`  
`sql-operator`

---

**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.

---

**sql-libraries**  
*Variable*

**Package**  
`sql`
**Initial value**

nil

**Description**

Holds a pathname or list of libraries to override default database library loading. The value should be a pathname or a list.

If its value is a pathname, it is prepended to a list of relative pathnames in the same manner that the supplied environment variable (for example `ORACLE_HOME`) would be. If its 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**

**Variable**

**Package**

sql

**Initial value**

nil

**Description**

The variable *sql-loading-verbose* controls the verbosity of messages while loading the database libraries.

**Notes**

*sql-loading-verbose* is applicable only on Unix.

---

**sql-operation**

**Function**

**Summary**

Generates a SQL statement from an operator and arguments.

**Package**

sql

**Signature**

sql-operation \( op \ & \ rest \ args \Rightarrow sql\-result \)

sql-operation \( sql\-function \ name \ & \ rest \ args \Rightarrow sql\-result \)

sql-operation \( sql\-operator \ inop1 \ left \ & \ rest \ rights \Rightarrow sql\-result \)
This chapter applies to the Enterprise Edition only

\[
\text{sql-operation \ sql-boolean-operator inop2 left &rest rights => sql-result}
\]

**Arguments**
- **op**
  - An operator.
- **args**
  - A set of arguments for \(\text{op}\).
- **name**
  - An arbitrary function.
- **args**
  - A set of arguments for \(\text{name}\).
- **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 \(\text{sql-operation}\) takes an operator and its arguments, and returns a SQL expression.

\[
(\text{sql-operation op args})
\]

is shorthand for

\[
(\text{apply (sql-operator op) args}).
\]

The pseudo operator \(\text{sql-function}\) allows an arbitrary function \(\text{name}\) to be passed. In this case, \(\text{name}\) is put in the SQL expression using \(\text{princ}\), and \(\text{args}\) are given as arguments.

The pseudo operators \(\text{sql-boolean-operator}\) and \(\text{sql-operator}\) generate SQL that calls an infix operator with \(\text{left}\) on the left and \(\text{rights}\) on the right separated by spaces. Use \(\text{sql-boolean-operator}\) for SQL infix operators that return a boolean and use \(\text{sql-operator}\) for any other SQL infix operator.
The pseudo operator `sql-operator` should not be confused with the Common SQL function `sql-operator`.

The following code, uses `sql-operation` to produce a SQL expression.

```scheme
(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`:

```scheme
(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`

---

### Function

**sql-operator**

**Summary**

Returns the symbol for a SQL operator.
This chapter applies to the Enterprise Edition only

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*. 
The SQL Package  This chapter applies to the Enterprise Edition only

See also

<table>
<thead>
<tr>
<th>Function</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql-stream</td>
<td>Returns the broadcast stream used for recording SQL commands or results traffic.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add-sql-stream</code></td>
</tr>
<tr>
<td><code>delete-sql-stream</code></td>
</tr>
<tr>
<td><code>list-sql-streams</code></td>
</tr>
<tr>
<td><code>sql-stream</code></td>
</tr>
<tr>
<td><code>start-sql-recording</code></td>
</tr>
<tr>
<td><code>stop-sql-recording</code></td>
</tr>
</tbody>
</table>
This chapter applies to the Enterprise Edition only

sql-recording-p
start-sql-recording
stop-sql-recording

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  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type</code></td>
<td>A keyword.</td>
</tr>
<tr>
<td><code>database</code></td>
<td>A database.</td>
</tr>
</tbody>
</table>

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 \textit{standard-output}.
You can modify the broadcast list of recording streams using \texttt{add-sql-stream} and \texttt{delete-sql-stream}.

type is one of \texttt{:commands}, \texttt{:results} or \texttt{:both}. It determines whether SQL commands traffic, results traffic or both is recorded.

The default value of type is \texttt{:commands}. The default value for database is the value of \texttt{*default-database*}.

See also \texttt{add-sql-stream} \texttt{delete-sql-stream} \texttt{list-sql-streams} \texttt{sql-stream} \texttt{sql-recording-p} \texttt{stop-sql-recording}

\textbf{status}

\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} 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}. 

\texttt{status}
The SQL Package

This chapter applies to the Enterprise Edition only

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
stop-sql-recording &key type database =>

Arguments

\begin{align*}
\text{type} & : \text{A keyword.} \\
\text{database} & : \text{A database.}
\end{align*}

Values
None.

Description
The function \textit{stop-sql-recording} stops recording SQL commands or results traffic.

\text{type} is one of \textit{commands}, \textit{results} or \textit{both}. It determines whether the recording of SQL commands traffic, results traffic or both is stopped.

The default value of \text{type} is \textit{commands}. The default value for \text{database} is \textit{*default-database*}.

See also
add-sql-stream
delete-sql-stream
list-sql-streams
sql-recording-p
sql-stream
start-sql-recording
table-exists-p  

**Function**

**Summary**  
A predicate for the existence of a table.

**Package**  
sql

**Signature**  
\[
\text{table-exists-p} \; \text{table} \; \&\text{key} \; \text{database} \; \text{owner} \Rightarrow \text{result}
\]

**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**

\[
\text{update-instance-from-records} \; \text{instance} \; \&\text{key} \; \text{database} \Rightarrow \text{instance}
\]
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**

*Function*

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`.
This chapter applies to the Enterprise Edition only

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.

`sloths` provides a list of the names of slots to update. Each of these slots should be a remote join slot (as defined above).

`sloths` can also be `t`, meaning update all the remote join slots. The default value of `sloths` is `t`.

`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**

```plaintext
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.

**Description** 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.

**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.

Package
`sql`

Signature
`update-slot-from-record instance slot => instance`

Arguments

- `instance` An instance of a View Class.
- `slot` A slot name.

Values
`instance` The updated View Class instance.

Description
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.
This chapter applies to the Enterprise Edition only

*instance* must be associated with a 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-instance-from-records

**with-transaction**

*Macro*

**Summary**
Performs a body of code within a transaction for a database.

**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.

**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*
The STREAM Package

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

Class

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
(example-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  

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  

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.
See also  
fundamental-binary-input-stream  
fundamental-binary-output-stream  
fundamental-stream  
stream-element-type

**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 340.

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
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 340.

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  
\texttt{fundamental-binary-input-stream}  
\texttt{fundamental-character-input-stream}  
\texttt{fundamental-stream}  
\texttt{input-stream-p}

\textbf{fundamental-output-stream}  
\textit{Class}

\textbf{Summary}  
A class whose inclusion causes \texttt{output-stream-p} to return \texttt{t}.

\textbf{Package}  
\texttt{stream}

\textbf{Superclasses}  
\texttt{fundamental-stream}

\textbf{Subclasses}  
\texttt{fundamental-binary-output-stream}  
\texttt{fundamental-character-output-stream}

\textbf{Description}  
The class \texttt{fundamental-output-stream} is a superclass to the binary and character output classes. Its inclusion causes the generic function \texttt{output-stream-p} to return \texttt{t}.

\textbf{See also}  
\texttt{fundamental-binary-output-stream}  
\texttt{fundamental-character-output-stream}  
\texttt{fundamental-stream}  
\texttt{input-stream-p}

\textbf{fundamental-stream}  
\textit{Class}

\textbf{Summary}  
A class whose inclusion causes \texttt{stremap} to return \texttt{t}.

\textbf{Package}  
\texttt{stream}

\textbf{Superclasses}  
\texttt{standard-object}  
\texttt{stream}
Subclasses  
- fundamental-binary-stream
- fundamental-character-stream
- fundamental-input-stream
- fundamental-output-stream

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*

Summary  
Writes the required number of blank spaces to ensure that the next character will be written in a given column.

Package  
`stream`

Signature  
`stream-advance-to-column stream column => result`

Arguments  
- `stream`  
  A stream.
- `column`  
  An integer.

Values  
- `result`  
  A boolean.

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.
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 342.

See also: fundamental-output-stream
stream-file-position

**Generic Function**

**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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>A stream.</td>
</tr>
<tr>
<td>position-spec</td>
<td>A file position designator.</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>A file position or nil.</td>
</tr>
<tr>
<td>success-p</td>
<td>A generalized boolean.</td>
</tr>
</tbody>
</table>

**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

**Generic Function**

**Summary**
Fills the stream buffer.

**Package**
stream
### stream-fill-buffer

**Signature**

\[ \text{stream-fill-buffer } \text{stream } \Rightarrow \text{result} \]

**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**

\[ \text{stream-finish-output } \text{stream } \Rightarrow \text{nil} \]
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 342.

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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream</code></td>
<td>An output stream.</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>result</code></td>
<td>A generalized boolean.</td>
</tr>
</tbody>
</table>

**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

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 342.

See also
fundamental-output-stream

stream-fresh-line

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 `buffered-stream` that handle input.

See also
- buffered-stream
- stream-fill-buffer

---

**stream-read-byte**

**Generic Function**

**Summary**
A generic function used by `read-byte` to read an integer or `:eof` symbol from a binary stream.

**Package**
`stream`

**Signature**
`stream-read-byte stream => result`

**Arguments**
- `stream` An input stream.

**Values**
- `result` An integer or `:eof`.

**Description**
The generic function `stream-read-byte` is used by `read-byte`, and returns either an integer read from the binary stream specified by `stream`, or the keyword `:eof`.

A method must be implemented for all binary subclasses of `buffered-stream` that handle input. A typical implementation will call `stream-read-char` and convert the character to an integer using `char-code`.

A method should be defined for a subclass of `fundamental-binary-input-stream`.

See also
- buffered-stream
- fundamental-binary-input-stream
- fundamental-binary-stream
- 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 \texttt{:eof} if end-of-file is reached, or \texttt{nil} if no input is available. The default method provided by \texttt{fundamental-character-input-stream} simply calls \texttt{stream-read-char} which is sufficient for file streams, but interactive streams should define their own method.

See also \texttt{fundamental-character-input-stream} \texttt{stream-read-char}

\begin{center}
\textbf{stream-read-line} \hfill \textit{Generic Function}
\end{center}

\textbf{Summary} \hfill Returns a string read from a stream.

\textbf{Package} \hfill \texttt{stream}

\textbf{Signature} \hfill \texttt{stream-read-line stream \Rightarrow result terminated}

\textbf{Arguments} \hfill \texttt{stream} \hfill An input stream.

\textbf{Values} \hfill \texttt{result} \hfill A string or \texttt{:eof}.

\texttt{terminated} \hfill A boolean.

\textbf{Description} \hfill The generic function \texttt{stream-read-line} reads a line of characters from \texttt{stream} and returns this line as a string. If the string is terminated by an end-of-file instead of a newline then \texttt{terminated} is \texttt{t}.

The default method uses repeated calls to \texttt{stream-read-char}, and uses \texttt{stream-element-type} to determine the element-type of its result.

See also \texttt{fundamental-character-input-stream} \texttt{stream-element-type} \texttt{stream-read-char}
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

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

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).
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 sub-class 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.

### 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`.

See also

- `buffered-stream`
- `stream-flush-buffer`
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 342.

See also `fundamental-character-output-stream`

### stream-write-sequence

**Generic Function**

**Summary**  
Writes a subsequence of a sequence to a stream.

**Package**  
`stream`
Signature

\texttt{stream-write-sequence stream sequence start end => result}

Arguments

\begin{itemize}
\item \texttt{stream} \quad A stream.
\item \texttt{sequence} \quad A sequence.
\item \texttt{start} \quad An integer.
\item \texttt{end} \quad An integer.
\end{itemize}

Values

\texttt{result} \quad A sequence.

Description

The generic function \texttt{stream-write-sequence} is used by \texttt{write-sequence} to write a subsequence of \texttt{sequence} delimited by \texttt{start} and \texttt{end} to \texttt{stream}.

A default method is provided by \texttt{fundamental-character-output-stream} that tests each element of \texttt{sequence} in turn, and then uses \texttt{stream-write-char} or produces an error. A default method is provided by \texttt{fundamental-binary-output-stream} that tests each element of \texttt{sequence} in turn, and then uses \texttt{stream-write-byte} or produces an error.

See also

\begin{itemize}
\item \texttt{fundamental-binary-output-stream}
\item \texttt{fundamental-character-output-stream}
\item \texttt{stream-read-sequence}
\item \texttt{stream-write-byte}
\item \texttt{stream-write-char}
\end{itemize}

\textbf{stream-write-string} \quad \textit{Generic Function}

Summary

Used by \texttt{write-string} to write a string to a character output stream.

Package

\texttt{stream}

Signature

\texttt{stream-write-string stream string &optional start end => result}
Arguments

- **stream**: A stream.
- **string**: A string.
- **start**: An integer.
- **end**: An integer.

Values

- **result**: A string.

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 342.

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

- `buffer, index, limit`: Variables.
  - `stream`: An input stream.
  - `body`: Code.

Values

- `result`: The value returned by `body`.
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**.
(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.
Description

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
The SYSTEM Package

This chapter describes symbols available in the SYSTEM package. Various uses of the symbols documented here are discussed throughout this manual.

**apply-with-allocation-in-gen-num**

*Function*

**Summary**

Allows control over which generation objects are allocated in, in 64-bit LispWorks.

**Package**

system

**Signature**

```lisp
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.
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`.

See also

`allocation-in-gen-num`
`make-array`
`*symbol-alloc-gen-num*`

approaching-memory-limit

Condition

Summary

The class of conditions signalled when 32-bit LispWorks approaches its memory limit.

Package

`syste`m

Superclasses

`storage-condition`

Subclasses

None.
The condition class `approaching-memory-limit` is used for signalling an error when 32-bit LispWorks approaches its memory limit.

`approaching-memory-limit` is not relevant to 64-bit LispWorks.

"Approaching the memory limit" on page 124

`set-approaching-memory-limit-callback`

**atomic-decf**

**atomic-incf**

**Macros**

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

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 253, or expand to one of them.

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 253, 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 def and incf, but does the operation atomically.

Package system
**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 253, 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**

*place* => *element*

### Summary

Like **pop**, but does the operation atomically.

### Package

system

### Signature

**atomic-pop**

### Arguments

*place* One of the specific set of places defined for low level atomic operations.

### Values

*element* An object.
**The SYSTEM Package**

**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 253, 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**

`atomic-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 253, 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

<table>
<thead>
<tr>
<th>Summary</th>
<th>Deprecated synonyms for <code>text-string</code> and <code>simple-text-string</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td><code>system</code></td>
</tr>
</tbody>
</table>
| 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` |

### Functions

**augmented-string-p**

**simple-augmented-string-p**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Deprecated synonyms for <code>text-string-p</code> and <code>simple-text-string-p</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td><code>system</code></td>
</tr>
</tbody>
</table>
| Signature | `augmented-string-p object => result`  
`simple-augmented-string-p object => result` |
| Arguments | `object` A Lisp object. |
| Values   | `result` A boolean. |
The functions `augmented-string-p` and `simple-augmented-string-p` are deprecated synonyms for `text-string-p` and `simple-text-string-p`.

See also `text-string-p`  
`simple-text-string-p`

*binary-file-type*  

Variable

Summary The default file type of binary files.

Package `system`

Initial value The initial value of `*binary-file-type*` depends on the host CPU and the LispWorks implementation. See “Naming conventions for FASL files” on page 570.

Description The variable `*binary-file-type*` is the file type that `load` and `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*`.

See also `*binary-file-types*`  
`load-data-file`

*binary-file-types*  

Variable

Summary A list of file types that are loaded as binary files.

Package `system`

Initial value `nil`
**Description**
The variable *binary-file-types* contains a list of strings naming file types which load and require recognize as binary (FASL) files. FASL files are the output of `compile-file`, `dump-forms-to-file` or `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 *binary-file-type*).

**See also**
*binary-file-type*

---

**call-system**

**Function**

**Package**

`system`

**Signature**

call-system command &key current-directory wait shell-type => status, signal-number

**Arguments**

- `command` A string, a list of strings, a simple-vector of strings, or nil.
- `current-directory` A string. Implemented only on Microsoft Windows.
- `wait` A boolean.
- `shell-type` A string or nil.

**Values**

- `status` The exit status of the invoked shell or process.
- `signal-number` On Unix-like systems, if the process was terminated by a signal this is the signal number, otherwise nil. Always nil on Microsoft Windows.

**Description**
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 \texttt{terminal-io} in LispWorks.)

If \texttt{command} is a string then it is passed to the shell as the command to run, using the \texttt{-c} option, without any other arguments. The type of shell to run is determined by \texttt{shell-type} as described below. Note that for typical Unix shells, the string \texttt{command} may contain multiple commands separated by \texttt{;} (semicolon).

If \texttt{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 \texttt{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 \texttt{command} is \texttt{nil}, then the shell is run.

On Microsoft Windows, if \texttt{command} is a string, LispWorks hides the first window of the execution of the command, because that is the console that \texttt{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 \texttt{current-directory} is the \texttt{lpCurrentDirectory} argument passed to \texttt{CreateProcess}. If this is not supplied, the \texttt{pathname-location} of the \texttt{current-pathname} is passed.

If \texttt{wait} is true, \texttt{call-system} does not return until the process has exited. The default for \texttt{wait} is \texttt{t}.
On Unix/Linux/Mac OS X/FreeBSD, 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. Some older unsupported versions of Windows use command.com instead.

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 400.

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 (or command.com) 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:

(call-system (format nil "adb ~a < ~a > ~a"  
 (namestring a)  
 (namestring b)  
 (namestring c)))

On Microsoft Windows:
(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 400

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 Unix/Linux/Mac OS X.
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 Unix/Linux/Mac OS X 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`. 
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 Unix/Linux/Mac OS X/FreeBSD, 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

Function

Summary
A generalized reference for alist elements.

Package
system

Signature
cdr-assoc item alist &key test test-not key => result
(setf cdr-assoc) value item alist => value

Arguments
item An object.
alist An association list.
test A function designator.
test-not A function designator.
key A function designator.
value An object.

Values
result An object (from alist) or nil.

Description
The functions cdr-assoc and (setf cdr-assoc) provide a
generalized reference for elements in an association list. The
arguments are all as specified for the Common Lisp function
assoc. cdr-assoc and (setf cdr-assoc) read and write
the cdr of an element in a manner consistent with the Com-
mon Lisp notion of places.

cdr-assoc returns the cdr of the first cons in the alist alist
that satisfies the test, or nil if no element of alist matches.

(setf cdr-assoc) modifies the first cons in alist that satis-
fies the test, setting its cdr to value. If no element of alist
matches, then (setf cdr-assoc) constructs a new cons
(cons item value) and inserts it in the head of alist.
Example

```lisp
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 or Macintosh.

coerce-to-gesture-spec

Function

Summary Returns a Gesture Spec object.

Package system

Signature `coerce-to-gesture-spec object &optional errorp => gspec`

Arguments `object` A character, keyword, Gesture Spec or string.
errorp  A boolean.

Values  gspec  A Gesture Spec object

Description  The function coerce-to-gesture-spec returns a Gesture Spec object gspec which can be used to represent the key-stroke indicated by object.

If object is a Lisp character, then gspec’s data is its cl:char-code and gspec’s modifiers are 0.

If object is a keyword, then it must be one of the known Gesture Spec keywords and becomes gspec’s data. gspec’s modifiers is 0.

If object is a string, then 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 name-char or the name of one of the known Gesture Spec keywords. Then gspec contains the corresponding Gesture Spec keyword or char-code in its data, and the modifier keys are represented in its modifiers.

If object is a Gesture Spec object, it is simply returned.

coerce-to-gesture-spec does not create wild gesture specs.

Examples

```lisp
(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
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.

Description
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 253, 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 registry 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.

**Package**

`system`

**Signature**

`count-gen-num-allocation gen-num &optional include-lower-generations`

**Arguments**

- `gen-num` An integer between 0 and 7, inclusive.
- `include-lower-generations` A generalized boolean.

**Values**

- `allocation` An integer.

**Description**

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.

See also room-values

**debug-initialization-errors-in-snap-shot** Variable

Summary Controls use of the snapshot debugger.

Package system

Initial value t

Description The variable *debug-initialization-errors-in-snap-shot* controls whether, in an image which is configured to 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-snap-shot* 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 UNIX/Linux/Mac OS X. 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 64KB 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.
- **lambda-list**
  A define-modify-macro lambda list.
- **function**
  A symbol.
- **doc-string**
  A string, not evaluated.

**Values**

- **name**
  A symbol.

**Description**

The macro define-atomic-modify-macro has the same syntax as cl:define-modify-macro, and performs a similar operation.

The resulting macro 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 253. The macro name reads the value of the place, calls the function
function, and then writes the result of the function call if the value in 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 function may be called more than once for each invocation of the defined macro. Therefore function should not have any side effects.

2. function must be thread-safe, because it may run concurrently in several threads if the defined macro name is used from several threads simultaneously.

3. It is possible in principle for the value to change more than once between reading the place and writing the new value. This may end up resetting the value in place to its original value, and hence the operation will succeed. This is equivalent to the code being invoked after the last change, unless function itself looks at place, which may cause inconsistent results.

See also 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 `result-type` is `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 `result-type` is `nil`, then the last form should return two values. If the second value is `nil` then the first value is treated as a list of values to returned to the top level loop. If the second 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:
(define-top-loop-command (:lave
    (:result-type eval)) (form)
  (reverse form))

then the command line

;lave (1 2 list)

will evaluate the form (list 2 1).

Here are definitions for two commands both of which will run apropos:

(define-top-loop-command (:apropos-eval
    (:result-type eval))
  (&rest args)
  ~(apropos ,@args))

(define-top-loop-command :apropos-noeval (&rest args)
  (apply 'apropos args))

The first one will evaluate the arguments before calling apropos whereas the second one will just pass the forms, so

;lave :apropos-noeval foo

will find all the symbols containing the string foo, whereas

(setq foo "bar")

;lave :apropos-eval foo

will find all the symbols containing the string bar.

---

detect-eol-style

Function

Summary
Detecteds 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

```lisp
(:default :eol-style :crlf)
```

If found and is not preceded by (13), merge ef-spec with

```lisp
(: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 (unsigned-byte 16), search buffer up to length for the first occurrence of the byte sequence (13 0 10). If found, merge ef-spec with

```lisp
(:default :eol-style :crlf)
```

If (13 0 10) is not found, search buffer up to length for (10 0) or (0 10). If found, merge ef-spec with

```lisp
(:default :eol-style :lf)
```

Thus a complete external format spec is constructed. Otherwise, return ef-spec.

**See also**  *file-eol-style-detection-algorithm*
detect-japanese-encoding-in-file

Function

Summary
Determines which type of Japanese encoding is used in a buffer.

Package
system

Signature
detect-japanese-encoding-in-file 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 Japanese encoding that was found.

Description
Assume the encoding is one of :jis, :sjis, :euc, :unicode and :ascii, and try to determine which of these it is, by looking for distinctive byte sequences in buffer up to length. If found, merge ef-spec with that encoding.

See also
*file-encoding-detection-algorithm*

detect-unicode-bom
detect-utf32-bom
detect-utf8-bom

Functions

Summary
Looks for the Unicode Byte Order Mark, which if found is assumed to indicate the matching Unicode encoding.

Package
system
Signature

detect-unicode-bom pathname ef-spec buffer length => new-ef-spec

detect-utf32-bom pathname ef-spec buffer length => new-ef-spec

detect-utf8-bom pathname ef-spec buffer length => new-ef-spec

Arguments

pathname Pathname identifying the 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

These functions are called as part of open's encoding detection routine, and try to detect the encoding if it is not already supplied in the 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 Unix-like systems, 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 256 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 256 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 256 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-loads-after-loads => nil`

**Description**
`ensure-loads-after-loads` is a synchronization function which ensures order of memory between operations in different threads.
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:
It also affects whitespace-char-p.

See also whitespace-char-p

**file-encoding-detection-algorithm**

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 locale-file-encoding)

Description 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.

Notes 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.

**Example**

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:

```lisp
(find-encoding-option locale-file-encoding)
```

There are further examples in “Guessing the external format” on page 386.

**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`
The SYSTEM Package

*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*  

**Variable**

**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  

**Function**

**Summary**
Examines a buffer for an encoding option.

**Package**
**system**
Signature

\texttt{find-encoding-option \ pathname \ ef-spec \ buffer \ length \ => \ result}

Arguments

- \texttt{pathname}: Pathname identifying 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{result}: The result of reading the value returned from the \texttt{encoding} or \texttt{external-format} option as a Lisp expression in the \texttt{keyword} package.

Description

Looks in the file options (EMACS-style \texttt{-**-} line) for an option called \texttt{encoding} or \texttt{external-format}, with value \texttt{value}.

If \texttt{encoding} or \texttt{external-format} is found, it reads \texttt{value} as a Lisp expression in the \texttt{keyword} package. If \texttt{coding} is found, it attempts to translate \texttt{value} from a GNU Emacs coding system name to a LispWorks external-format name.

It then merges the \texttt{ef-spec} with the external format spec derived from \texttt{value}, and returns the result as \texttt{result}. Thus it does not override a supplied \texttt{ef-spec}.

See also

\texttt{*file-encoding-detection-algorithm*}

\textbf{find-filename-pattern-encoding-match} \quad \textit{Function}

Summary

Finds the encoding of a file based on the filename.

Package

\texttt{system}

Signature

\texttt{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.
The function `gen-num-segments-fragmentation-state` shows the fragmentation state in a generation in 64-bit LispWorks.

`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 `set-blocking-gen-num` and `marking-gc`).

Allocation types `:cons-static`, `:non-pointer-static`, `:mixed-static`, `:other-big` and `:non-pointer-big` are included in the result only if `statics-too` is non-nil. The default value of `statics-too` is nil.

Notes

1. The implementation of `set-blocking-gen-num` is intended to solve any fragmentation issues automatically.

2. `gen-num-segments-fragmentation-state` is implemented only in 64-bit LispWorks. It is not relevant to the Memory Management API in 32-bit implementations, where `check-fragmentation` is available instead.

See also

- `check-fragmentation`
- `marking-gc`
- `set-blocking-gen-num`

“Guidance for control of the storage management system” on page 112
generation-number

Function

Summary
Returns the current generation number for an object.

Package
system

Signature
generation-number object => integer

Arguments
object
A Lisp object.

Values
integer
An integer.

Description
The function generation-number returns the generation number in which the Lisp object object currently is. See the discussion in Chapter 11, “Storage 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 storage management system” on page 112

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.
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
- `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` 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, **gesturespec-to-character** signals an error.

See also

- **coerce-to-gesture-spec**
- **make-gesture-spec**

---

**get-file-stat**

**Function**

**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.
The device type (sometimes called Rdev).

**get-folder-path**  
*Function*

**Summary**  
Gets the path of a special folder.

**Package**  
`system`

**Signature**  
`get-folder-path what &key create => result`

**Arguments**  
`what` A keyword.  
`create` A boolean.

**Values**  
`result` A directory pathname naming the path, or `nil`.

**Description**  
The function `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. `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 `<homedir>/get-folder-path/<symbol-name-down-cased>`. This allows testing code that uses `get-folder-path` to work in the sense that files can be written and read from these directories.

`what` indicates the purpose of the special folder. For instance, `:common-appdata` means the folder containing application data for all users.
The following values of what are recognized on Microsoft Windows and Mac OS X:


:documents is an alias for :my-documents.

The following values are recognized on Microsoft Windows only: :program-files, :programs and :common-programs.

The following values are recognized on Mac OS X only:


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 keyword), 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:

(make-pathname
 :name "foo"
 :type "lisp"
 :defaults
 (sys:get-folder-path :my-documents))
See also \texttt{get-user-profile-directory}

\begin{description}
\item[get-user-profile-directory] \textit{Function}
\item[Summary] Gets the root of the user’s profile on a Windows NT-based system.
\item[Package] \texttt{system}
\item[Signature] \texttt{get-user-profile-directory} $\Rightarrow$ \texttt{result}
\item[Values] \texttt{result} \hspace{1em} A directory pathname naming the path, or \texttt{nil}.
\item[Description] The function \texttt{get-user-profile-directory} obtains the path to the current user’s profile folder on a Windows NT-based system (including Windows Vista, Windows 7 and Windows 8). \texttt{get-user-profile-directory} is implemented only on Microsoft Windows.
\item[result] names the root of the profile directory.
\item[Compatibility notes] In LispWorks 5.0 and previous versions, \texttt{get-user-profile-directory} returns a string.
\item[Example] On Windows 8 and Windows 7:
\begin{verbatim}
(sys:get-user-profile-directory) => #P"C:/Users/dubya/"
\end{verbatim}
On Windows XP (now unsupported):
\begin{verbatim}
(sys:get-user-profile-directory) => #P"C:/Documents and Settings/dubya/"
\end{verbatim}
\end{description}
On Windows 98 SE (now unsupported):

```
(sys:get-user-profile-directory)
=>
nil
```

See also `get-folder-path`

---

**guess-external-format**

*Function*

**Summary**
Tries to work out the external format

**Package**
`system`

**Signature**
`guess-external-format pathname ef-spec buffer length => 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**
- `ef-spec` An external format spec.

**Description**
If `ef-spec` is complete, then it is returned. Otherwise `guess-external-format` calls, in turn, functions on the list `*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 `:default`, an error of type `file-encoding-resolution-error` is signaled. The caller offers a restart for trying again with respecified `external-format` and/or `element-type` arguments. Otherwise `guess-external-format` proceeds to guess the `eol-style`. 
To guess the *eol-style*, functions on the list *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 :eol-style, an error of type file-encoding-resolution-error is signaled.

See also

*file-encoding-detection-algorithm*
*file-eol-style-detection-algorithm*
file-encoding-resolution-error

### immediatep

**Function**

**Summary**
The predicate for immediate objects.

**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*
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.

Example  

(system:in-static-area (make-string 10))

See also  

enlarge-static  

make-array  

staticp

int32

A type used to generate optimal 32-bit arithmetic code.

Package  

system

Signature  

int32

Description  

The type int32 is used to generate optimal 32-bit arithmetic code.

Objects of type 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 int32 objects to produce efficient raw 32-bit code.
See the section “Fast 32-bit arithmetic” on page 410 for more information.

See also

- int32*
- +int32-0+
- +int32-1+
- int32-1+
- int32/= 
- int32<<
- int32-aref
- int32-logand
- int32-minusp
- int32-to-integer
- integer-to-int32
- make-simple-int32-vector
- simple-int32-vector

### Functions

#### Summary

The arithmetic operators 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.

Description 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.
The function int32/ is the divide operator for int32 objects.
See the section “Fast 32-bit arithmetic” on page 410 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 \texttt{+int32-1+} expands to \texttt{(sys:integer-to-int32 1)}.

See also  \texttt{integer-to-int32}

\textbf{int32-1+  \hspace{1cm} int32-1-}

\textit{Functions}

Summary  The operators for \texttt{int32} objects corresponding to the functions \texttt{1+} and \texttt{1-}.

Package  \texttt{system}

Signatures  \begin{align*}
\texttt{int32-1+} \ x & \Rightarrow \texttt{int32} \\
\texttt{int32-1-} \ x & \Rightarrow \texttt{int32}
\end{align*}

Arguments  \begin{align*}
x & \text{ An \texttt{int32} object or an integer of type } \\
& \phantom{=} \texttt{(signed-byte 32).}
\end{align*}

Values  \begin{align*}
\texttt{int32} & \text{ An \texttt{int32} object.}
\end{align*}

Description  The functions \texttt{int32-1+} and \texttt{int32-1-} are the operators for \texttt{int32} objects corresponding to the functions \texttt{1+} and \texttt{1-}.

See the section “Fast 32-bit arithmetic” on page 410 for more information about the INT32 API.

See also  \texttt{int32}
Functions

Summary

The comparison operators for int32 objects.

Package

system

Signatures

int32/= x y => result
int32< x y => result
int32<= x y => result
int32= x y => result
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 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 410 for more information about the INT32 API.

See also `int32`

### `int32<<`  
### `int32>>`

**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 410 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.
x An int32 object or an integer of type (signed-byte 32).

Values

int32 An int32 object.

Description

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 410 for more information about the INT32 API.

See also

int32

simple-int32-vector
The bitwise logical operators for int32 objects.

Package

system

Signatures

int32-logand x y => int32
int32-logandc1 x y => int32
int32-logandc2 x y => int32
int32-logbitp index x => result
int32-logeqv x y => int32
int32-logior x y => int32
int32-lognand x y => int32
int32-lognor x y => int32
int32-lognot x => int32
int32-logorc1 x y => int32
int32-logorc2 x y => int32
int32-logtest x y => result
\texttt{int32-logxor \ x \ y \ => \ int32}

**Arguments**

\begin{itemize}
\item \texttt{x} \quad \text{An int32 object or an integer of type (signed-byte 32).}
\item \texttt{y} \quad \text{An int32 object or an integer of type (signed-byte 32).}
\item \texttt{index} \quad \text{An int32 object or an integer of type (signed-byte 32).}
\end{itemize}

**Values**

\begin{itemize}
\item \texttt{int32} \quad \text{An int32 object.}
\item \texttt{result} \quad \text{An boolean.}
\end{itemize}

**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 \texttt{x} with \texttt{y}.

The function \texttt{int32-logandc2} is the bitwise logical operator for \texttt{int32} objects which `ands' \texttt{x} with the complement of \texttt{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 \(x\).

The function \texttt{int32-logorc1} is the bitwise logical operator for \texttt{int32} objects which ‘inclusive ors’ the complement of \(x\) with \(y\).

The function \texttt{int32-logorc2} is the bitwise logical operator for \texttt{int32} objects which ‘inclusive ors’ \(x\) with the complement of \(y\).

The function \texttt{int32-logtest} is the bitwise test for \texttt{int32} objects which returns \texttt{t} if any of the bits designated by \texttt{1} in \(x\) is \texttt{1} in \(y\), 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 410 for more information about the INT32 API.

\textbf{See also} \hspace{1em} \texttt{int32}

\begin{description}
  \item[\texttt{int32-minusp}] \hspace{1em} \texttt{x => result}
  \item[\texttt{int32-plusp}] \hspace{1em} \texttt{x => result}
  \item[\texttt{int32-zerop}] \hspace{1em} \texttt{x => result}
\end{description}

\textbf{Functions}

\textbf{Summary} \hspace{1em} The \texttt{minusp}, \texttt{plusp} and \texttt{zerop} tests for an \texttt{int32} object.

\textbf{Package} \hspace{1em} \texttt{system}

\textbf{Signatures} \hspace{1em} \texttt{int32-minusp \hspace{1em} x => result}
\texttt{int32-plusp \hspace{1em} x => result}
\texttt{int32-zerop \hspace{1em} x => result}

\textbf{Arguments} \hspace{1em} \texttt{x} \hspace{1em} An \texttt{int32} object or an integer of type \texttt{(signed-byte 32)}. 

\textbf{1590}
### int32-to-int64

**Function**

**Summary**
Converts from int32 to int64.

**Package**
system

**Signature**
int32-to-int64 x => y

**Arguments**

- x : An int32 object.

**Values**

- y : An int64 object.

**Description**

The function int32-to-int64 converts the int32 object x to the corresponding int64 object y.

**See also**

- int32
- int64
- int64-to-int32

---

### Values

<table>
<thead>
<tr>
<th>result</th>
<th>A boolean.</th>
</tr>
</thead>
</table>

### 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 410 for more information about the INT32 API.

**See also**

- int32
**int32-to-integer**  
*Function*

**Summary**  
The destructor converting an int32 object to an integer.

**Package**  
*system*

**Signature**  
```
int32-to-integer int32 => integer
```

**Arguments**  
`int32`  
An `int32` object or an integer of type (signed-byte 32).

**Values**  
`integer`  
An integer of type (signed-byte 32).

**Description**  
The function `int32-to-integer` returns an integer `integer` of type (signed-byte 32) corresponding to the int32 object `int32`. The argument `int32` can also be an integer of type (signed-byte 32), in which case it is simply returned.

An error is signaled if `int32` is not of type `int32` or (signed-byte 32).

See the section “Fast 32-bit arithmetic” on page 410 for more information about the INT32 API.

**See also**  
`int32`

---

**int64**  
*Type*

**Summary**  
A type used to generate optimal 64-bit arithmetic code.

**Package**  
*system*

**Signature**  
```
int64
```

**Description**  
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 412 for more information.

See also

int64* +int64-0+ +int64-1+ int64-1+ int64/= int64<< int64-aref int64-logand int64-minusp int64-to-integer integer-to-int64 make-simple-int64-vector simple-int64-vector

### Functions

**Summary**
The arithmetic operators for int64 objects.

**Package**

system

**Signatures**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int64* x y =&gt; int64</td>
<td></td>
</tr>
<tr>
<td>int64+ x y =&gt; int64</td>
<td></td>
</tr>
<tr>
<td>int64- x y =&gt; int64</td>
<td></td>
</tr>
<tr>
<td>int64/ x y =&gt; int64</td>
<td></td>
</tr>
</tbody>
</table>
Arguments

\( x \)  
An \texttt{int64} object or an integer of type \texttt{(signed-byte 64)}.

\( y \)  
An \texttt{int64} object or an integer of type \texttt{(signed-byte 64)}.

Values

\texttt{int64}  
An \texttt{int64} object.

Description

The function \texttt{int64*} is the multiply operator for \texttt{int64} objects.

The function \texttt{int64+} is the add operator for \texttt{int64} objects.

The function \texttt{int64-} is the subtract operator for \texttt{int64} objects.

The function \texttt{int64/} is the divide operator for \texttt{int64} objects.

See the section “Fast 64-bit arithmetic” on page 412 for more information about the INT64 API.

See also

\texttt{int64}

\texttt{+int64-0+}  
\textit{Symbol Macro}

Summary

Shorthand for \texttt{(sys:integer-to-int64 0)}.

Package  
\texttt{system}

Description

The symbol macro \texttt{+int64-0+} expands to \texttt{(sys:integer-to-int64 0)}.

See also

\texttt{integer-to-int64}

\texttt{+int64-1+}  
\textit{Symbol Macro}

Summary

Shorthand for \texttt{(sys:integer-to-int64 1)}.
Package system

Description The symbol macro +int64-1+ expands to (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 int64-1+ x => int64
int64-1- x => int64

Arguments x An int64 object or an integer of type (signed-byte 64).

Values int64 An int64 object.

Description The functions int64-1+ and int64-1- are the operators for int64 objects corresponding to the functions 1+ and 1-.

See the section “Fast 64-bit arithmetic” on page 412 for more information about the INT64 API.

See also int64
Functions

The comparison operators for int64 objects.

Package

system

Signatures

int64/= x y => result
int64< x y => result
int64<= x y => result
int64= x y => result
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 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 412 for more information about the INT64 API.

See also `int64`

### `int64<<`  
### `int64>>`

**Summary**  
The shift operators for `int64` objects.

**Package**  
`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 412 for more information about the INT64 API.

See also int64

**int64-aref**  
*Function*

**Summary**  
The accessor for a simple-int64-vector.

**Package**  
*system*

**Signature**  
```lisp
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 index in vector. The writer sets the value at index index in vector to the int64 object or integer x supplied.

See the section “Fast 64-bit arithmetic” on page 412 for more information about the INT64 API.

See also int64  
simple-int64-vector
int64-logand
int64-logandc1
int64-logandc2
int64-logbitp
int64-logeqv
int64-logior
int64-lognand
int64-lognor
int64-lognot
int64-logorc1
int64-logorc2
int64-logtest
int64-logxor

Summary
The bitwise logical operators for int64 objects.

Package
system

Signatures
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
int64-logior x y => int64
int64-lognand x y => int64
int64-lognor x y => int64
int64-lognot x => int64
int64-logorc1 x y => int64
int64-logorc2 x y => int64
int64-logtest x y => result
The SYSTEM Package

\[ \text{int64-logxor } x \ y = \text{int64} \]

**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).
- **index**: An int64 object or an integer of type (signed-byte 64).

**Values**

- **int64**: An int64 object.
- **result**: An boolean.

**Description**

The function \text{int64-logand} is the bitwise logical 'and' operator for \text{int64} objects.

The function \text{int64-logandcl} is the bitwise logical operator for \text{int64} objects which 'ands' the complement of \text{x} with \text{y}.

The function \text{int64-logandc2} is the bitwise logical operator for \text{int64} objects which 'ands' \text{y} with the complement of \text{x}.

The function \text{int64-logbitp} is the test for \text{int64} objects which returns \text{t} if if the bit at index \text{index} in \text{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 \text{x} and \text{y}.

The function \text{int64-logior} is the bitwise logical 'inclusive or' operator for \text{int64} objects.

The function \text{int64-lognand} is the bitwise logical operator for \text{int64} objects which returns the complement of the 'and' of \text{x} and \text{y}.

The function \text{int64-lognor} is the bitwise logical operator for \text{int64} objects which returns the complement of the 'inclusive or' of \text{x} and \text{y}.
The function `int64-lognot` is the bitwise logical operator for `int64` objects which returns the complement of its argument \( x \).

The function `int64-logorc1` is the bitwise logical operator for `int64` objects which 'inclusive ors' the complement of \( x \) with \( y \).

The function `int64-logorc2` is the bitwise logical operator for `int64` objects which 'inclusive ors' \( x \) with the complement of \( y \).

The function `int64-logtest` is the bitwise test for `int64` objects which returns \( t \) if any of the bits designated by 1 in \( x \) is 1 in \( y \), and returns \( \text{nil} \) otherwise.

The function `int64-logxor` is the bitwise logical 'exclusive or' operator for `int64` objects.

See the section “Fast 64-bit arithmetic” on page 412 for more information about the INT64 API.

See also `int64`

### Functions

**int64-minusp**

**int64-plusp**

**int64-zerop**

**Summary**

The `minusp`, `plusp` and `zerop` tests for an `int64` object.

**Package**

`system`

**Signatures**

`int64-minusp x => result`

`int64-plusp x => result`

`int64-zerop x => result`

**Arguments**

\( x \)

An `int64` object or an integer of type `(signed-byte 64)`.
Values

Result: A boolean.

Description

The function `int64-minusp` tests whether its argument $x$ is \texttt{int64}< than the value of \texttt{+int64-0+}.

The function `int64-plusp` tests whether its argument $x$ is \texttt{int64}> than the value of \texttt{+int64-0+}.

The function `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 412 for more information about the INT64 API.

See also `int64`

\textbf{int64-to-int32}

\textit{Function}

Summary

Converts from \texttt{int64} to \texttt{int32}.

Package

\texttt{system}

Signature

\texttt{int64-to-int32 $x \Rightarrow y$}

Arguments

$x$ An \texttt{int64} object.

Values

$y$ An \texttt{int32} object.

Description

The function \texttt{int64-to-int32} converts the \texttt{int64} object $x$ to the corresponding \texttt{int32} object $y$.

See also `int32`

\texttt{int32-to-int64}

\texttt{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 412 for more information about the INT64 API.

See also  int64

---

### integer-to-int32

**Function**

**Summary**  The constructor for int32 objects.

**Package**  system

**Signature**  integer-to-int32 integer => int32

**Arguments**  integer  An integer of type (signed-byte 32).

**Values**  int32  An int32 object.
The function `integer-to-int32` constructs an `int32` object from an integer. An error is signaled if `integer` is not of type `(signed-byte 32)`.

See the section “Fast 32-bit arithmetic” on page 410 for more information about the INT32 API.

See also `int32`

---

**integer-to-int64**

*Function*

**Summary** The constructor for `int64` objects.

**Package** system

**Signature** `integer-to-int64 integer => int64`

**Arguments** `integer` An integer of type `(signed-byte 64)`.

**Values** `int64` An `int64` object.

**Description** The function `integer-to-int64` constructs an `int64` object from an integer. An error is signaled if `integer` is not of type `(signed-byte 64)`.

See the section “Fast 64-bit arithmetic” on page 412 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 392.

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 in LispWorks 7.0

View the full manual entry for details: load-data-file.

locale-file-encoding  Function

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
The **SYSTEM Package**

**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*
- *multibyte-code-page-ef*
- safe-locale-file-encoding

---

**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 253.

See also  
atomic-decf
atomic-exchange
atomic-fixnum-decf
atomic-pop
atomic-push
compare-and-swap
define-atomic-modify-macro

make-gesture-spec  
Function

Summary  
Create a Gesture Spec object.

Package  
system

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.
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
- :f11
- :f12
- :f13
- :f14
- :f15
- :f16
- :f17
:f18
:f19
:f20
:f21
:f22
:f23
:f24
:f25
:f26
:f27
:f28
:f29
:f30
:f31
:f32
:f33
:f34
:f35
:help
:left
:right
:up
:down
:home
: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
:insert-line
:delete-line
:insert-char
:delete-char
:prev-item
:next-item
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-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:

```lisp
(sys:make-gesture-spec
  (char-code #\x)
  sys:gesture-spec-shift-bit)
```

Instead you should use:

```lisp
(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-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 \textit{initial-contents}, if supplied, should be a sequence of length \textit{length}. It specifies the contents of \textit{vector}.

The argument \textit{initial-element}, if supplied, specifies the contents of \textit{vector}.

An error is signaled if both \textit{initial-contents} and \textit{initial-element} are supplied.

See the section “Fast 32-bit arithmetic” on page 410 for more information about the INT32 API.

\textbf{See also} \texttt{int32} \texttt{simple-int32-vector}

\begin{knitrout}

\begin{itemize}
  \item \textbf{make-simple-int64-vector} \textit{Function}
  \item \textbf{Summary} The constructor for \texttt{simple-int64-vector} objects.
  \item \textbf{Package} \texttt{system}
  \item \textbf{Signature} \texttt{make-simple-int64-vector length \\&key initial-contents initial-element }\Rightarrow \texttt{vector}
  \item \textbf{Arguments} \begin{itemize}
    \item \textit{length} A non-negative fixnum.
    \item \textit{initial-contents} A sequence of integers of type (\texttt{signed-byte 64}), or \texttt{nil}.
    \item \textit{initial-element} An integer of type (\texttt{signed-byte 64}).
  \end{itemize}
  \item \textbf{Values} \begin{itemize}
    \item \textit{vector} A \texttt{simple-int64-vector}.
  \end{itemize}
  \item \textbf{Description} The function \texttt{make-simple-int64-vector} is the constructor for \texttt{simple-int64-vector} objects.
    
    The argument \textit{initial-contents}, if supplied, should be a sequence of length \textit{length}. It specifies the contents of \textit{vector}.
\end{itemize}
\end{knitrout}
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 412 for more information about the INT64 API.

See also int64
simple-int64-vector

make-stderr-stream

Function

Returns an output stream connected to stderr.

Summary

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 runtime 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.

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 410

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
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 \([0, 7]\).

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.

See also gc-generation
set-blocking-gen-num
“Guidance for control of the storage management system” on page 112

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

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 117

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.
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.

---

**object-address**

*Function*

**Summary**

Returns the address of a Lisp object.

**Package**

system

**Signature**

object-address object => address

**Arguments**

object A Lisp object.

**Values**

address An integer.

**Description**

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.
See also  
  immediatep  
  object-pointer  
  pointer-from-address

**object-pointer**  

*Function*

Summary  Returns an integer specifying the representation of an object.

Package  system

Signature  object-pointer object => result

Arguments  
  object  
  A Lisp object.

Values  
  result  
  An integer.

Description  

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 "immediate" (that is they do not use memory, and **immediatep** returns t) and for these **object-pointer** returns the actual address.

The Garbage Collector can move objects, therefore the result of **object-pointer** is not permanent. It should be used only for debugging.

Notes  

The result of **object-pointer** is what **cl:print-unreadable-object** uses for the object’s "identity". It is normally what appears when using **cl:print-unreadable-object** with identity t.

Examples  

```
(let ((gf #'make-instance))
  (format t "-a pointer is -x-%" gf
    (sys:object-pointer gf)))
```
See also  
  immediatep  
  object-address  
  pointer-from-address

**octet-ref**

**base-char-ref**

**Functions**

**Summary**

Loads an octet from a simple vector and returns it as an integer or base-char.

**Package**

system

**Signature**

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

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector</td>
<td>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.</td>
</tr>
<tr>
<td>octet-index</td>
<td>A non-negative integer.</td>
</tr>
<tr>
<td>int</td>
<td>An integer.</td>
</tr>
<tr>
<td>char</td>
<td>A base-char.</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iresult</td>
<td>An integer in the inclusive range [0, 255].</td>
</tr>
<tr>
<td>cresult</td>
<td>A base-char.</td>
</tr>
</tbody>
</table>

**Description**

The functions octet-ref and base-char-ref 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 410

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.
direction :input, :output, :io or :none.
element-type A type specifier.
interrupt-off A boolean. Not implemented on Microsoft Windows.
shell-type A shell type.
use-pty A boolean.
save-exit-status A boolean.

Values  
stream A pipe stream.

Description  
On Unix/Linux/Mac OS X/FreeBSD 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". Note that on Windows ME/98/95 you will need to pass "command".

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:
The following example shows you how to use bidirectional pipes.

Example on Microsoft Windows
CL-USER 40 > (setf *ls* (sys:open-pipe "dir"))
#<WIN32::TWO-WAY-PIPE-STREAM 205F03F4>

CL-USER 41 > (loop while
  (print (read-line *ls* nil nil)))

" Volume in drive Z is lispsrc"
" Volume Serial Number is 82E3-1342"**
" Directory of Z:\v42\delivery-tests"
"20/02/02 11:57a        <DIR>          ."
"20/02/02 11:57a        <DIR>          .."
"14/02/02 07:04p             6,815,772 othello.exe"
"14/02/02 07:07p             6,553,628 hello.exe"
"                          4 File(s)     13,369,400 bytes"
"                          3,974,103,040 bytes free"
NIL
NIL

CL-USER 42 > (close *ls*)
T

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:

(setq *sub*
  (sys:open-pipe
   (list (lisp-image-name)
     "-eval"
     "(quit :status 1623)"
     :save-exit-status t))

This form then returns 1623:

(sys:pipe-exit-status *sub*)

See also
call-system
call-system-showing-output
pipe-exit-status
pipe-kill-process
open-url

Function

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

Function

Summary Queries whether a package is flagged.

Package system

Signature package-flagged-p package flag => result

Arguments package A package designator.
flag A keyword.
Values

result: A boolean.

Description

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-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 before calling `close`.

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 400 for the interpretation of the exit status and the signal number.

See also:
- `open-pipe`
- `call-system`
- “Interpreting the exit status” on page 400

**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 => gspec
```

**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 keystroke 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`

*print-symbols-using-bars*  

Variable

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 readable case and `*print-case*` are both `:upcase` or both `:downcase`.  

1635
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)
(setf product-registry-path) path product => path
```

**Arguments**

- `product`: A Lisp object.
- `path`: The path as a string or a list of strings.

**Values**

- `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 Unix/Linux/Mac OS X 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`

**room-values**

**Function**

**Summary**
Returns information about the state of internal storage.
package  system

signature  room-values => result

values  result  A plist

::total-size  size
::total-allocated  allocated
::total-free  free

description  room-values returns a plist containing information about
the state of internal storage. 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 storage management system” on
page 112

run-shell-command  function

package  system

summary  Allows executables and DOS or Unix shell commands to be
called from Lisp code.

signature  run-shell-command command &key input output error-output
separate-streams wait if-input-does-not-exist if-output-exists if-error-output-exists show-window environment element-type save-exit-status => result

signature  run-shell-command command &key input output error-output
separate-streams wait if-input-does-not-exist if-output-exists if-error-output-exists show-window environment element-type save-exit-status => stream, error-stream, process
**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>A string, a list of strings, a simple-vector of strings, or nil.</td>
</tr>
<tr>
<td>input</td>
<td>nil, :stream or a file designator. Default value nil.</td>
</tr>
<tr>
<td>output</td>
<td>nil, :stream or a file designator. Default value nil.</td>
</tr>
<tr>
<td>error-output</td>
<td>nil, :stream, :output or a file designator. Default value nil.</td>
</tr>
<tr>
<td>separate-streams</td>
<td>A boolean. True value not currently supported.</td>
</tr>
<tr>
<td>wait</td>
<td>A boolean, default value t.</td>
</tr>
<tr>
<td>if-input-does-not-exist</td>
<td>:error, :create or nil. Default value :error.</td>
</tr>
<tr>
<td>show-window</td>
<td>A boolean. True value not currently supported.</td>
</tr>
<tr>
<td>environment</td>
<td>An alist of strings naming environment variables and values. Default value nil.</td>
</tr>
<tr>
<td>element-type</td>
<td>Default value base-char.</td>
</tr>
<tr>
<td>save-exit-status</td>
<td>A boolean, default value nil.</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>The exit status of the process running command, or a process ID</td>
</tr>
<tr>
<td>stream</td>
<td>A stream, or nil.</td>
</tr>
<tr>
<td>error-stream</td>
<td>A stream, or nil.</td>
</tr>
</tbody>
</table>
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 `wait` is `nil` and `error-output` has the value `:stream` then `run-shell-command` executes `command` and returns three values. `stream` is determined by the arguments `input` and `output` as described above. `error-stream` is a Lisp stream which acts as the stderr of the process. `process` is the process ID of the process.

If `wait` is `nil` and `error-output` is not `:stream` then the second return value, `error-stream`, is `nil`. If `error-output` is `:output`, then stderr goes to the same place as stdout.

If `input` is a pathname or string, then `open` is called with `:if-does-not-exist` if `input-does-not-exist`. The resulting `file-stream` acts as the stdin of the process.

If `output` is a pathname or string, then `open` is called with `:if-exists` if `output-exists`. The resulting `file-stream` acts as the stdout of the process.

If `error-output` is a pathname or string, then `open` is called with `:if-exists` if `error-output-exists`. The resulting `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><code>input</code> is <code>:stream</code></td>
<td>An I/O stream connected to stdin and stdout</td>
<td>An input stream connected to stderr</td>
</tr>
<tr>
<td><code>output</code> is <code>:stream</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>error-output</code> is <code>:stream</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>input</code> is not <code>:stream</code></td>
<td>An input stream connected to stdout</td>
<td>An input stream connected to stderr</td>
</tr>
<tr>
<td><code>output</code> is <code>:stream</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>error-output</code> is <code>:stream</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>input</code> is <code>:stream</code></td>
<td>An output stream connected to stdin</td>
<td>An input stream connected to stderr</td>
</tr>
<tr>
<td><code>output</code> is not <code>:stream</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>error-output</code> is <code>:stream</code></td>
<td></td>
<td></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 <code>:stream</code></td>
<td>nil</td>
<td>An input stream connected to stderr</td>
</tr>
<tr>
<td><code>output</code> is not <code>:stream</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>error-output</code> is <code>:stream</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>input</code> is <code>:stream</code></td>
<td>An I/O stream connected to stdin, stdout and stderr</td>
<td>nil</td>
</tr>
<tr>
<td><code>output</code> is <code>:stream</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>error-output</code> is <code>:output</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>input</code> is not <code>:stream</code></td>
<td>An input stream connected to stdout and stderr</td>
<td>nil</td>
</tr>
<tr>
<td><code>output</code> is <code>:stream</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>error-output</code> is <code>:output</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>input</code> is <code>:stream</code></td>
<td>An output stream connected to stdin</td>
<td>nil</td>
</tr>
<tr>
<td><code>output</code> is not <code>:stream</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>error-output</code> is <code>:stream</code> or <code>:output</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>input</code> is not <code>:stream</code></td>
<td>An I/O stream connected to stdin and stdout</td>
<td>nil</td>
</tr>
<tr>
<td><code>output</code> is <code>:stream</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>error-output</code> is not <code>:stream</code> or <code>:output</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>input</code> is not <code>:stream</code></td>
<td>An input stream connected to stdout</td>
<td>nil</td>
</tr>
<tr>
<td><code>output</code> is not <code>:stream</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>error-output</code> is not <code>:stream</code> or <code>:output</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>input</code> is not <code>:stream</code></td>
<td>An output stream connected to stdin</td>
<td>nil</td>
</tr>
<tr>
<td><code>output</code> is not <code>:stream</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>error-output</code> is not <code>:stream</code> or <code>:output</code></td>
<td></td>
<td></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
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:

```
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 124
- `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**

```
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 128
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:
49  The SYSTEM Package

\[ t \] 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 \textit{fragmentation-threshold} (the same as the argument to \texttt{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 \texttt{do-gc} is \( t \).

\textit{max-size-to-copy} is meaningful only if \texttt{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 \texttt{gc-threshold} is non-nil, it is used to set the threshold for automatic GC using \texttt{set-gen-num-gc-threshold}.

The initial setup is as if this call has been made:

\[
\text{(sys:set-blocking-gen-num 3)}
\]

That is, the system will GC automatically according to the default \texttt{gc-threshold} using Copying GC.

Setting the blocking generation \texttt{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 \texttt{gen-num} during the operation. The macro \texttt{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 `gc-generation` explicitly to promote long living objects to a higher generation. The advantage of doing this is that you can call `gc-generation` in places where you know there are not many short-lived objects alive.

Passing a `do-gc` value other than `t` is useful when the blocking generation can be large enough that copying it all may cause very serious paging. Passing `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 `gc-generation` or `marking-gc`.

`set-blocking-gen-num` returns four values: the old blocking generation number, the old value of `do-gc`, the `max-size-to-copy`, and the old value of `gc-threshold`. It can be called with `gen-num nil` to query the values without changing any of them.

Notes

`set-blocking-gen-num` is implemented only in 64-bit LispWorks. It is not relevant to the Memory Management API in 32-bit implementations.

See also

`block-promotion`  
`gc-generation`  
`marking-gc`  
`set-automatic-gc-callback`  
`set-gen-num-gc-threshold`  
“Guidance for control of the storage management system” on page 112

**set-default-segment-size**

*Function*

**Summary**

Sets the default initial size of a segment in 64-bit LispWorks.
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.
set-default-segment-size is implemented only in 64-bit LispWorks. 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 128

set-delay-promotion

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.

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 Lisp-Works. It is not relevant to the Memory Management API in 32-bit implementations.

See also

set-blocking-gen-num

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-gen-num-gc-threshold

**Summary**
Sets the additional allocation threshold that triggers a GC in the blocking generation in 64-bit LispWorks.

**Package**
*system*

**Signature**

```
set-gen-num-gc-threshold gen-num threshold => old-threshold
```

**Arguments**

- **gen-num**
  An integer between 0 and 7, inclusive.
- **threshold**
  An integer greater than 12800, or a real in the inclusive range [0 100], or nil.

**Values**

- **old-threshold**
  A number.

**Description**

The function `set-gen-num-gc-threshold` sets the threshold for additional allocation that triggers a garbage collection (GC) in generation `gen-num` when this is the blocking generation (as set by `set-blocking-gen-num`). A GC is triggered when the allocation in generation `gen-num` grows more than `threshold` over the allocation after the last GC of this generation (or a GC of a higher generation).

To set the threshold, `threshold` can be an integer greater than 12800, which is interpreted as the absolute value. Alternatively `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 `gen-num`, when generation `gen-num` is the blocking generation and allocation in it has doubled since the previous GC, generation `gen-num` is collected automatically.

`set-gen-num-gc-threshold` can be called when the generation `gen-num` is not the blocking generation, and will set the value for that `gen-num`. Such a call will not take effect until the generation `gen-num` becomes the blocking generation, as
set by a call to `set-blocking-gen-num` (with `: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 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 128

---

**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.

In 32-bit implementations on platforms other than Linux and Macintosh, by default the maximum memory is not set. LispWorks (32-bit) for Linux and LispWorks (32-bit) for Macintosh both set 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 \texttt{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 \texttt{memory-growth-margin} or \texttt{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 \texttt{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 \texttt{check-fragmentation}  
\texttt{mark-and-sweep}  
\texttt{memory-growth-margin}  
\texttt{room-values}  
“Memory Management in 32-bit LispWorks” on page 117

\textbf{set-maximum-segment-size} 

\textit{Function}

\textbf{Summary} Defines the maximum segment size for a generation and allocation type in 64-bit LispWorks.

\textbf{Package} \texttt{system}

\textbf{Signature} \texttt{set-maximum-segment-size gen-num allocation-type size-in-mb}

\textbf{Arguments}  
\texttt{gen-num} An integer between 0 and 7, inclusive.  
\texttt{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 129.

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 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 128

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.
set-memory-exhausted-callback

**Function**

Sets a callback that is called when memory is exhausted in 64-bit LispWorks.

**Package**

system

**Signature**

set-memory-exhausted-callback function &optional where => callbacks

**Arguments**

*function* A function designator, the keyword :reset, or nil.

*where* :first, :last or nil.

**Values**

*callbacks* A list of function designators.

**Description**

The function set-memory-exhausted-callback adds a callback that is called when memory is exhausted. That is, when the system fails to map memory.

*Note:* set-memory-check is a more robust way to protect against memory exhaustion problems.

If *function* is a function designator then it should be a function with signature

function gen-num size type-name static

*function* is expected to report what the system was trying to allocate when it failed to map memory. Its arguments are:

*gen-num* The number of the generation in which it was trying to allocate.

set-memory-check is implemented only in 64-bit LispWorks. It is not relevant to the Memory Management API in 32-bit implementations.

See also set-memory-exhausted-callback
The size in bytes which it was trying to allocate.

A string naming the allocation type it was trying to allocate.

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.
set-signal-handler

Summary
Installs or removes a handler for a Unix signal.

Note: applicable only on UNIX/Linux/Mac OS X.

Package
system

Signature
set-signal-handler signum handler

Arguments
signum A Unix signal number.
handler A function or nil.

Description
set-signal-handler with a function handler configures LispWorks such that handler is called when the Unix 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 :bug-form listener command. For example, there is a SIGINT handler which calls break. You should consult Lisp Support before overwriting existing signal handlers.
LispWorks initially has no SIGHUP handler. SIGHUP will kill a LispWorks process which does not have a SIGHUP handler installed. When the LispWorks IDE starts up, a SIGHUP handler (which attempts to release locks in the environment) is installed. However if you need a SIGHUP handler in a server application, for example, you should install one using `set-signal-handler`.

```
(defun my-hup-handler (&rest x)
  (declare (ignorable x))
  (cerror "Continue"
         "Got a HUP signal")

(sys:set-signal-handler 1 'my-hup-handler)
```

Note that the LispWorks IDE overwrites a SIGHUP handler, so you would need to reinstall it after GUI startup.

### set-spare-keeping-policy

**Function**

**Summary**

Controls the behavior of the system when a segment is emptied in 64-bit LispWorks.

**Package**

`system`

**Signature**

`set-spare-keeping-policy gen-num policy => old-policy`

**Arguments**

- `gen-num` An integer in the inclusive range [0,7].
- `policy` A generalized boolean.

**Values**

- `old-policy` A generalized boolean.

**Description**

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`.

Notes

`set-spare-keeping-policy` is implemented only in 64-bit LispWorks. It is not relevant to the Memory Management API in 32-bit implementations.

See also `extended-time`

---

**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`. 
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

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

\begin{verbatim}
(apply (car function-and-arguments)
       (cdr function-and-arguments))
\end{verbatim}

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 
\texttt{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.

\texttt{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.

\texttt{setup-atomic-funcall} causes less congestion than using a 
lock, and so is more efficient for locks that may cause conge-
tion. \texttt{compare-and-swap} and \texttt{atomic-exchange} operations 
will be faster.

See also \texttt{atomic-exchange} \texttt{compare-and-swap}

\textbf{*sg-default-size*} \textit{Variable}

\begin{tabular}{|l|}
\hline
\textbf{Summary}  & Default initial size of a stack group. \\
\hline
\textbf{Package}  & \texttt{system} \\
\hline
\textbf{Initial value}  & In LispWorks (64-bit) for Solaris: 20000 \\
                        & In all other implementations: 16000 \\
\hline
\textbf{Description} & The value of the variable \texttt{*sg-default-size*} is the initial 
                        size of a stack group, in 32 bit words (in 32-bit implemen-
                        tations) or in 64 bit words (in 64-bit implementations). 
                        
                        \texttt{*sg-default-size*} can be bound around a call to a process 
                        creation function. Note that setting the global value of this 
\end{tabular}
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*

### simple-int32-vector

**Type**

**Summary**
A type for simple vectors of `int32` objects.

**Package**
`system`

**Signature**
simple-int32-vector

**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 410 for more information.

See also  
`int32`  
`int32-aref`  
`make-simple-int32-vector`

### simple-int64-vector

**Type**

**Summary**
A type for simple vectors of `int64` objects.
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 412 for more information.

See also `int64`  
`int64-aref`  
`make-simple-int64-vector`

*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 86l). 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*

**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.
- :type A string naming the allocation type the system was trying to allocate.
- :static 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-condition](#) 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

- \texttt{sweep-gen-num-objects} returns \texttt{nil}.

Description

The function \texttt{sweep-gen-num-objects} applies \texttt{function} to all the live objects in the generation \texttt{gen-num}.

\texttt{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

\texttt{sweep-gen-num-objects} is not implemented in 32-bit LispWorks, where you can use \texttt{sweep-all-objects} instead.

See also \texttt{sweep-all-objects}

---

**typed-aref**

*Function*

**Summary**

Accesses a typed aref vector efficiently.

**Package**

\texttt{system}

**Signature**

\texttt{typed-aref type vector byte-index => value}

\texttt{(setf typed-aref) value type vector byte-index => value}

**Arguments**

- \texttt{type}: A type specifier.
- \texttt{vector}: A vector created by \texttt{make-typed-aref-vector}.
- \texttt{byte-index}: A non-negative fixnum.

**Values**

- \texttt{value}: An object of type \texttt{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 \texttt{fli:foreign-typed-aref} in the \textit{LispWorks Foreign Language Interface User Guide and Reference Manual}

Example

\begin{verbatim}
(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)
\end{verbatim}

See also \texttt{make-typed-aref-vector}

“Optimized integer arithmetic and integer vector access” on page 410

\begin{flushleft}
\textbf{wait-for-input-streams} \quad \textit{Function}
\end{flushleft}

\textbf{Summary}

Waits for input on a list of socket streams, returning those that are ready.

\textbf{Package} \quad \texttt{system}

\textbf{Signature}

\texttt{wait-for-input-streams arguments \&key wait-function wait-reason timeout \=> result}

\textbf{Arguments}

- \texttt{streams} \quad A list, each member of which is a \texttt{socket-stream}.
- \texttt{wait-function} \quad A function of no arguments.
- \texttt{wait-reason} \quad A string.
- \texttt{timeout} \quad A real number or \texttt{nil}.

\textbf{Values}

- \texttt{result} \quad A list of \texttt{socket-streams} or \texttt{nil}.

\textbf{Description}

The function \texttt{wait-for-input-streams} waits for any of the streams in the argument \texttt{streams} to be ready for input. “Ready for input” typically means that some input is available from the stream, but can also means that the peer closed the con-
nection 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, `wait-for-input-streams` returns a list of all the streams that are ready,
in the same order that they appear in `streams`.

If `timeout` is non-nil it must be a real number, specifying a timeout in seconds. If `timeout` seconds pass and none of the
streams is ready, `wait-for-input-streams` returns `nil`.

If `timeout` is 0, `wait-for-input-streams` returns all of the
streams that are ready immediately, without waiting at all.
That is, it behaves like `listen` on many streams.

If `wait-function` is supplied, it is called periodically with no arguments, and if it returns non-nil then `wait-for-input-
streams` returns `nil`. Note that, like the `wait-function` of `process-wait`, `wait-function` is called often and on other threads,
so need to be an inexpensive call and independent of dynamic context.

If `wait-reason` is supplied it is used as the `wait-reason` for the
Lisp process that calls `wait-for-input-streams` while it is waiting.

Notes `wait-for-input-streams` may return the list `streams` that
was passed to it as is, if all the streams are ready.

See also `wait-for-input-streams-returning-first`

---

### wait-for-input-streams-returning-first

**Function**

**Summary**

Waits for input on a list of socket streams, returning the first
stream that is ready.

**Package** `system`
**wait-for-input-streams-returning-first**

**Signature**

```lisp
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**

```lisp
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 255 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**  
```lisp
(with-modification-check-macro macro-name modification-place &body body)
```

**Arguments**  
modification-placeA 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 255 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 inside 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`
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 or Mac OS X platforms.

\textbf{canonicalize-sid-string} \quad \textit{Function}

\begin{tabular}{|l|}
\hline
Summary & Returns the canonical format of a SID specifier string. \\
Package & win32 \\
Signature & \texttt{canonicalize-sid-string \ sid-string \Rightarrow result} \\
\hline
\end{tabular}
Arguments  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid-string</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 `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`

**connect-to-named-pipe**  

**Function**

Summary  
Opens a stream connection to a named pipe.

Package  
win32

Signature  
`connect-to-named-pipe name &key host errorp direction`

Arguments  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A string.</td>
</tr>
<tr>
<td>host</td>
<td>A string or nil.</td>
</tr>
<tr>
<td>errorp</td>
<td>A boolean.</td>
</tr>
<tr>
<td>direction</td>
<td>One of the keywords :io, :input and :output.</td>
</tr>
</tbody>
</table>

Values  

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>A stream or nil.</td>
</tr>
<tr>
<td>keyword</td>
<td>A keyword or nil.</td>
</tr>
<tr>
<td>condition</td>
<td>An error condition or nil.</td>
</tr>
</tbody>
</table>
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.
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  known-sid-integer-to-sid-string integer => sid-string

Arguments  integer        An integer.

Values      sid-string    A string or nil.

Description The function known-sid-integer-to-sid-string returns the SID string for integer, which needs to be one of the known integers, which are documented in the MSDN in the entry for WELL_KNOWN_SID_TYPE Enumeration.

known-sid-integer-to-sid-string returns nil for an unknown integer.

See also  wait-for-connection
          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  win32

Initial value  (1252 28591)

Description The value of *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 (: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

**Function**

**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**

**Summary** Holds the external format corresponding to the current Windows multi-byte code page.

**Package** win32

**Description** 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.

**See also** locale-file-encoding

---

**named-pipe-stream-name**  

**Function**

**Summary** Returns the name of a named pipe stream.

**Package** win32

**Signature** named-pipe-stream-name stream => name

**Arguments**  

- **stream** A named pipe stream.

**Values**  

- **name** A string.

**Description** 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.

**See also** wait-for-connection
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`

---

### record-message-in-windows-event-log

**Summary**
Records a message in the Windows event log.

**Package**
`win32`
This chapter applies only to LispWorks for Windows

**Signature**

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".
This chapter applies only to LispWorks for Windows

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
**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 nil, *fail-type* is a short string giving the type of the item that fails, and *fail-item* is the item in the list that fails when *access-spec* is a list.
Notes

1. When the argument is syntactically incorrect, `security-description-string-for-open-named-pipe` signals an error. It fails and returns `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, `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 `:disallow-write` or `:disallow-read` is used, when it generates an ACE string denying permission. All the ACEs are then concatenated and "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 `open-named-pipe-stream`, the keywords `:disallow-read` etc are not very useful. They are useful only when you want to deny access for a specific SID, by using `:disallow-read` and `: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 `: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`. 

1695
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
- dismiss-splash-screen
- user-name-to-sid-string

**set-application-themed**

Function

Summary

Controls whether LispWorks should be themed.

Package

win32

Signature

set-application-themed on/off

Arguments

on/off A generalized boolean.

Description

The function **set-application-themed** controls whether a LispWorks application should be themed.
On supported versions of Microsoft Windows, LispWorks is "themed", that is it uses the current theme of the desktop. You can switch this off by calling

\[(\text{win32: set-application-themed nil})\]

On systems older than Windows XP, or when the application does not have Common Controls 6, this call has no effect.

\textit{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, \textit{set-application-themed} can be called multiple times in the same run.

---

**short-namestring**

\textbf{Function}

\textbf{Summary} 
Returns the short form of a namestring.

\textbf{Package} 
\texttt{win32}

\textbf{Signature} 
\texttt{short-namestring pathname => result}

\textbf{Arguments} 
\textit{pathname} 
A pathname designator.

\textbf{Values} 
\textit{result} 
A string or \texttt{nil}.

\textbf{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 \texttt{result} is a string in the short form.

The translation may fail, in which case \texttt{nil} is returned.

\textbf{See also} 
\texttt{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

**Signature**
str &key length
lpcstr &key max-length
lpstr &key max-length

**Description**
str is an ANSI string.
lpcstr 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.

See also tstr

tstr
lpctstr
lptstr

Summary
Types which automatically switch between ANSI and Unicode strings.

Package
win32

Signature

tstr &key length
lpctstr &key max-length
lptstr &key max-length

Description

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.

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:
(fli:define-foreign-function (%get-drive-type
"GetDriveType" :dbs)
  ((lpRootPathName W:LPCTSTR))
  :result-type (:unsigned :int))

(defconstant +drive-types+
  #'(:unknown :none :removable :fixed :remote :cdrom
    :ramdisk))

(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  tstr
This chapter applies only to LispWorks for Windows

**user-name-to-sid-string**

*Function*

**Summary**

Returns a standard SID (Security Identifier) associated with the user.

**Package**

`win32`

**Signature**

`user-name-to-sid-string user-name => sid-string`

**Arguments**

`user-name` A string.

**Values**

`sid-string` A string or `nil`.

**Description**

The function `user-name-to-sid-string` returns a standard SID (Security Identifier) associated with the user `user-name` on the current machine. It returns `nil` if it failed to find the user.

**See also**

`wait-for-connection`

`security-description-string-for-open-named-pipe`

**wait-for-connection**

*Generic Function*

**Summary**

Waits to establish a connection for a stream.

**Package**

`win32`

**Signature**

`wait-for-connection stream &key timeout wait-reason wait-function => connectedp`

**Arguments**

`stream` 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`.
<table>
<thead>
<tr>
<th>Values</th>
<th>connectedp</th>
<th>A generalized boolean.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>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).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>timeout can be nil or a real number specifying the time in seconds before wait-for-connection returns without establishing a connection.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wait-function, if non-nil, must be a function of no arguments. If it returns non-nil, wait-for-connection returns nil.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wait-for-connection can be repeatedly called on the same stream. If the stream has already established a connection, it returns true immediately.</td>
<td></td>
</tr>
<tr>
<td>Notes</td>
<td>wait-function has the same limitations as the wait-function of process-wait.</td>
<td></td>
</tr>
<tr>
<td>See also</td>
<td>open-named-pipe-stream</td>
<td></td>
</tr>
</tbody>
</table>

**with-windows-event-log-event-source**

**Macro**

**Summary**

Provides an open event log handle for a body of code.

**Package**

win32
This chapter applies only to LispWorks for Windows

**Signature**

\[
\text{with-windows-event-log-event-source} \ (\text{handle} \ \text{source-name} \ \&\text{optional} \ \text{unc-server-name}) \ \&\text{body} \ \text{body} \Rightarrow \text{values}
\]

**Arguments**

- **handle**
  - A symbol.
- **source-name**
  - \text{nil} or a string.
- **unc-server-name**
  - \text{nil} or a string.

**Values**

- **values**
  - The values returned by \text{body}

**Description**

The macro \text{with-windows-event-log-event-source} provides an open event log handle for a body of code.

The macro \text{with-windows-event-log-event-source} binds \text{handle} to an open event log handle, evaluates the forms of \text{body} and closes \text{handle}. The values of the last form in \text{body} are returned.

\text{source-name} is used as the name of the event source for recording events. If \text{source-name} is \text{nil} then the name of the Lisp executable is used.

If \text{unc-server-name} is non-nil, then it specifies the UNC name of a server which records the events.

**See also**

\text{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 or Mac OS X platforms.

### close-registry-key

**Function**

**Summary**

Closes a handle to an open registry key.

**Package**

`win32`
Signature  
\texttt{close-registry-key} \texttt{handle} \& \texttt{key} \texttt{errorp} \Rightarrow \texttt{successp}, \texttt{error-code}

Arguments  
\texttt{handle} \quad \text{A handle to an open registry key.}

Values  
\texttt{successp} \quad \text{A boolean.}
\texttt{error-code} \quad \text{An integer error code or \texttt{nil}.}

Description  
The function \texttt{close-registry-key} closes \texttt{handle}, which should be an open registry key handle.

The return value on success is \texttt{t}.

If an error occurs and \texttt{errorp} is true then an error is signaled. Otherwise, the return values are \texttt{nil} and the Windows \texttt{error-code}. The default value of \texttt{errorp} is \texttt{t}.

See also  
\texttt{create-registry-key}
\texttt{open-registry-key}

\textbf{collect-registry-subkeys}  
\textit{Function}

Summary  
Returns names of the subkeys of a registry key.

Package  
\texttt{win32}

Signature  
\texttt{collect-registry-subkeys} \texttt{subkey} \& \texttt{key} \texttt{root} \texttt{max-name-size} \texttt{max-names} \texttt{errorp} \texttt{value-function} \Rightarrow \texttt{subsubkeys}

Arguments  
\texttt{subkey} \quad \text{A string specifying the name of the key.}
\texttt{root} \quad \text{A keyword or handle.}
\texttt{max-name-size} \quad \text{An integer.}
\texttt{max-names} \quad \text{An integer.}
\texttt{errorp} \quad \text{A boolean.}
\texttt{value-function} \quad \text{A function designator or \texttt{nil}.}
This chapter applies only to LispWorks for Windows

Values

| subsubkeys | 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

$value-function handle subkey-name => name, collectp$

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

Function

collect-registry-values

Summary

Returns the values of a registry key.
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><code>collect-registry-values subkey &amp;key root max-name-size max-buffer-size expected-type errorp value-function =&gt; values-alist</code></td>
</tr>
</tbody>
</table>
| 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 subsubkey-name-and-value => name-and-value, collectp

value-function is funcalled for each subsubkey with the handle of subkey and a cons of the name and value of the subsubkey. 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 subkey 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 subkey &key class root access 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.
errorp A generalized boolean.

Values
handle The handle of the new key.
disposition A keyword, either :created-new-key or :opened-existing-key.
create-registry-key

**Description**

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 `:opened-existing-key` indicating whether a new key was created or opened.

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`

---

**delete-registry-key**

**Function**

**Summary**

Deletes a registry key.
This chapter applies only to LispWorks for Windows

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

Signature  

enum-registry-value subkey index &key root max-name-size
max-buffer-size expected-type errorp => name, data-type, data,
error-code

Arguments  

subkey  A string specifying the name of the key.
The Windows registry API

This chapter applies only to LispWorks for Windows

- **index** An integer.
- **root** A keyword or handle.
- **max-name-size** An integer.
- **max-buffer-size** An integer.
- **expected-type** A keyword or `t`.
- **errorp** A boolean.

**Values**

- **name** A string.
- **data-type** A keyword.
- **data** A lisp object.
- **error-code** An integer error code or `nil`.

**Description**

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 read-from-string</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
This chapter applies only to LispWorks for Windows

**query-registry-key-info**

*Function*

**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

Summary
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
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

```lisp
(registry-value subkey name &key root expected-type errorp =>
data, successp, error-code)
```

```lisp
(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
This chapter applies only to LispWorks for Windows

**set-registry-value**

**Function**

Summary
Stores a value in the registry.

Package
win32

Signature

\[
\text{set-registry-value} \; \text{data} \; \text{subkey} \; \text{name} \; \&\text{key} \; \text{root} \; \text{expected-type} \\
\text{errorp} \Rightarrow \text{error-code}
\]

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 51.2 Conversion of Lisp objects to registry values

<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 data</td>
</tr>
<tr>
<td>Lisp value</td>
<td>:lisp-object</td>
<td>REG_SZ made with prin1-to-string of data</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. The default value of `errorp` is `t`.

**See also**
- `create-registry-key`
- `registry-value`

### 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.
This chapter applies only to LispWorks for Windows

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
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 or Mac OS X platforms.

### dde-advice-start

**Function**

**Summary**

Sets up an advise loop on a specified data item for a conversation.
The function **dde-advise-start** sets up an advise loop for the data item specified by **item** on the specified **conversation**.

See “Advise loops” on page 278 for information about DDE advise loops.

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.
- 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 – the default value of **format**. 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 argument \textit{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 \texttt{: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 \textit{type} is specified as \texttt{:foreign}, which can be used with any clipboard format. It returns a \texttt{clipboard-item} structure, containing a foreign pointer to the data, the data length, and the format identifier.

If \textit{datap} is \texttt{t} (the default value), a hot link is established, where the new data is supplied whenever it changes. If \textit{datap} is \texttt{nil}, a warm link is established, where the data is not passed, and must be explicitly requested using \texttt{dde-request}.

The argument \textit{key} is used to identify this link. If specified as \texttt{nil} (the default value), it defaults to the conversation. Multiple links are permitted on a conversation with the same \textit{item} and \textit{format} values, as long as their \textit{key} values differ.

If the link is established, the return value \textit{result} is \texttt{t}. If the link could not be established, the behavior depends on the value of \texttt{errorp}. If \texttt{errorp} is \texttt{t} (the default value), LispWorks signals an error. If it is \texttt{nil}, the function returns \texttt{nil} to indicate failure.

If the link is established, the function \textit{function} is called whenever the data changes. If \texttt{function} is \texttt{nil} (the default value), then the generic function \texttt{dde-client-advise-data} will be called.

The function specified by \textit{function} should have a lambda list similar to the following:

\begin{verbatim}
key item data &key conversation &allow-other-keys
\end{verbatim}

The arguments \texttt{key} and \texttt{item} identify the link. The argument \texttt{data} contains the new data for hot links; for warm links it is \texttt{nil}. 

1725
See also
dde-advise-start*
dde-advise-stop
dde-client-advice-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-advice-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-advice-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-advice-start`
- `dde-advice-stop`
- `dde-advice-stop*`
- `dde-client-advice-data`

### dde-advice-stop

**Function**

**Summary**

Removes a link from a conversation specified by a given `item` and `key`. 
Package win32

Signature dde-advise-stop conversation item &key key format errorp disconnectp no-advise-ok => result

Arguments
- conversation: A conversation object.
- item: A string or symbol.
- key: An object.
- format: A clipboard format specifier.
- errorp: A boolean.
- disconnectp: A boolean.
- no-advise-ok: A boolean.

Values
- result: A boolean.

Description
The function dde-advise-stop removes a particular link from conversation specified 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 278 for information about DDE advise loops.

If disconnectp is t, and the last advise loop for the conversation is terminated, the conversation is disconnected.

Attempting to remove a link that does not exist raises an error, unless no-advise-ok is t.

If this function succeeds, it returns t. If it fails, the behavior depends on the value of errorp. If errorp is t (the default value), LispWorks signals an error. If errorp is nil, the function returns nil to indicate failure.

See also dde-advise-start
dde-advise-start*
This chapter applies only to LispWorks for Windows

\texttt{dde-advise-stop*}  
\texttt{dde-client-advise-data}

**dde-advise-stop***  

\textit{Function}

\textbf{Summary}  
Removes a link from an automatically managed conversation specified by a given item and key.

\textbf{Package}  
\texttt{win32}

\textbf{Signature}  
\texttt{dde-advise-stop* service topic item &key key format errorp disconnectp => result}

\textbf{Arguments}  
- \textit{service} A string or symbol.
- \textit{topic} A string or symbol.
- \textit{item} A string or symbol.
- \textit{key} An object.
- \textit{format} A clipboard format specifier.
- \textit{errorp} A boolean.
- \textit{disconnectp} A boolean.

\textbf{Values}  
- \textit{result} A boolean.

\textbf{Description}  
The function \texttt{dde-advise-stop*} is similar to the function \texttt{dde-advise-stop}, and removes a particular link from a conversation specified by the \textit{service/topic} pair indicated by \textit{item}, \textit{format} and \textit{key}. If \textit{key} is the last key for the \textit{item/format} pair, the advise loop for the pair is terminated.

See “Advise loops” on page 278 for information about DDE advise loops.

If \textit{disconnectp} is \texttt{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}
\texttt{dde-advise-start*}
\texttt{dde-advise-stop}

\textbf{dde-client-advise-data} \hspace{1cm} \textit{Generic Function}

Summary
Called when data changes in an advise loop.

Package \hspace{1cm} \texttt{win32}

Signature
\texttt{dde-client-advise-data key item data &key &allow-other-keys}

Arguments
\begin{itemize}
  \item \textit{key} \hspace{1cm} An object.
  \item \textit{item} \hspace{1cm} A string or symbol.
  \item \textit{data} \hspace{1cm} A string.
\end{itemize}

Values
None.

Description
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 key is used.

See “Advise loops” on page 278 for information about DDE advise loops.

See also
\texttt{dde-advise-start}
\texttt{dde-advise-stop}
This chapter applies only to LispWorks for Windows

**Function**

**dde-connect**

**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

\texttt{command} \hspace{1cm} \text{A string or symbol.}
\texttt{args} \hspace{1cm} \text{An argument.}

\textbf{Values}
\texttt{result} \hspace{1cm} \text{A boolean.}

\textbf{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.

\textbf{See also}
\texttt{dde-execute*} \\
\texttt{dde-execute-command*} \\
\texttt{dde-execute-string}

\texttt{dde-execute*} \hspace{1cm} \textit{Function}

\textbf{Summary}
An alternative syntax for \texttt{dde-execute-command*}.

\textbf{Package} \hspace{1cm} \texttt{win32}

\textbf{Signature}
\texttt{dde-execute* service topic command \&rest \{args\}* => result}

\textbf{Arguments}
\texttt{service} \hspace{1cm} \text{A string or symbol.}
\texttt{topic} \hspace{1cm} \text{A string symbol.}
\texttt{command} \hspace{1cm} \text{A string or symbol.}
\texttt{args} \hspace{1cm} \text{An argument.}

\textbf{Values}
\texttt{result} \hspace{1cm} \text{A boolean.}

\textbf{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

```
[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.
This chapter applies only to LispWorks for Windows

See also  
**dde-execute**  
**dde-execute-string**

dde-execute-command*  
*Function*

<table>
<thead>
<tr>
<th>Summary</th>
<th>Sends a command string to a specified service on a given topic.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>win32</td>
</tr>
<tr>
<td>Signature</td>
<td>dde-execute-command* service topic command arg-list &amp;key errorp connect-error-p new-conversation-p =&gt; result</td>
</tr>
<tr>
<td>Arguments</td>
<td></td>
</tr>
<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>
<tr>
<td>Values</td>
<td>result</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`
This chapter applies only to LispWorks for Windows

**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 277. 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 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.
This chapter applies only to LispWorks for Windows

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`, \texttt{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 \texttt{dde-execute} \linebreak \texttt{dde-execute-command} \linebreak \texttt{dde-execute-string}

\textbf{dde-item \hspace{1cm} Accessor}

Summary An accessor which can perform a request transaction or a poke transaction.

Package \texttt{win32}

Signature \texttt{dde-item conversation item &key format type errorp => result}

Arguments

\begin{itemize}
  \item \texttt{conversation} A conversation object.
  \item \texttt{item} A string or symbol.
  \item \texttt{format} A clipboard format specifier.
  \item \texttt{type} A keyword.
  \item \texttt{errorp} A boolean.
\end{itemize}
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

\begin{verbatim}
dde-item* service topic item &key format type errorp connect-error-p new-conversation-p => result
\end{verbatim}

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

\begin{verbatim}
(dde-request* service topic item :format format :type type :errorp errorp connect-error-p new-conversation-p)
\end{verbatim}

can also be issued using `dde-item*` as follows:

\begin{verbatim}
(dde-item* service topic item :FORMAT format :TYPE type :ERRORP errorp connect-error-p new-conversation-p)
\end{verbatim}

Similarly, the following `dde-poke*` command

\begin{verbatim}
(dde-poke* conversation item data :format format :type type :errorp errorp connect-error-p new-conversation-p)
\end{verbatim}

can be issued using `dde-item*` as follows:

\begin{verbatim}
(setf (dde-item* conversation item :format format :type type :errorp errorp connect-error-p new-conversation-p) data)
\end{verbatim}
except that the format always returns data.

If necessary, the accessor \texttt{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 \texttt{with-dde-conversation} prevents a new conversation being established for each transaction.

If \texttt{new-conversation-p} is set to \texttt{t} a new conversation is always established for the transaction. This new conversation is always automatically disconnected when the transaction is completed.

If \texttt{connect-error-p} is \texttt{t} (the default value), then LispWorks signals an error if a conversation cannot be established. If it is \texttt{nil}, \texttt{dde-item*} returns \texttt{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 \texttt{t}. On failure, the behavior depends on the value of the \texttt{errorp} argument. If \texttt{errorp} is \texttt{t} (the default value), LispWorks signals an error. If it is \texttt{nil}, the function returns \texttt{nil} to indicate failure.

See also \texttt{dde-item} \texttt{dde-poke} \texttt{dde-request}

\begin{description}
\item [dde-poke] \textbf{Function}
\item [Summary] Issues a poke transaction on a conversation, to set the value of a specified item.
\item [Package] \texttt{win32}
\end{description}
This chapter applies only to LispWorks for Windows

Signature

\texttt{dde-poke conversation item data &key format type errorp => result}

Arguments

- \textit{conversation} A conversation object.
- \textit{item} A string or symbol.
- \textit{data} A string.
- \textit{format} A clipboard format specifier.
- \textit{type} A keyword.
- \textit{errorp} A boolean.

Values

- \textit{result} A boolean.

Description

The function \texttt{dde-poke} issues a poke transaction on \textit{conversation} to set the value of the item specified by \textit{item} to the value specified by \textit{data}. The argument \textit{item} should be a string, or a symbol. If it is a symbol its print name is used.

The argument \textit{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 \textit{type} indicates that \textit{data} is a Lisp string to be used. If \textit{type} is :\texttt{string-list}, then \textit{data} is taken to be a list of strings, and is sent as a tab-separated string.
Alternatively, \texttt{data} can be a \texttt{clipboard-item} structure, containing a foreign pointer to the data to send and the length of the data. In this case the \texttt{type} argument is ignored.

On success, this function returns \texttt{t}. On failure, the behavior depends on the value of the \texttt{errorp} argument. If \texttt{errorp} is \texttt{t} (the default value), LispWorks signals an error. If it is \texttt{nil}, the function returns \texttt{nil} to indicate failure.

See also \texttt{dde-item} \texttt{dde-request}

\textbf{dde-poke*} \textit{Function}

\textbf{Summary} Issues a poke transaction on an automatically managed conversation, to set the value of a specified item.

\textbf{Package} \texttt{win32}

\textbf{Signature} \texttt{dde-poke* service topic item data &key format type errorp connect-error-p new-conversation-p => result}

\textbf{Arguments} \begin{itemize}
  \item \texttt{service} A symbol or string.
  \item \texttt{topic} A symbol or string.
  \item \texttt{item} A string or symbol.
  \item \texttt{data} A string.
  \item \texttt{format} A clipboard format specifier.
  \item \texttt{type} A keyword.
  \item \texttt{errorp} A boolean.
  \item \texttt{connect-error-p} A boolean.
  \item \texttt{new-conversation-p} A boolean.
\end{itemize}
Values

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>A boolean.</td>
</tr>
</tbody>
</table>

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 `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**

```lisp
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.
• 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*
Function

dde-request*

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.
The \textit{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 \textit{format}, \textit{type}, and \textit{errorp} arguments see \texttt{dde-request}.

If necessary, the function \texttt{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 \textit{service} and \textit{topic}, placing them inside a \texttt{with-dde-conversation} prevents a new conversation being established for each transaction.

If \texttt{new-conversation-p} is set to \texttt{t} a new conversation is always established for the transaction. This new conversation is always automatically disconnected when the transaction is completed.

If \texttt{connect-error-p} is \texttt{t} (the default value), then LispWorks signals an error if a conversation cannot be established. If it is \texttt{nil}, \texttt{dde-request*} returns \texttt{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.

\textbf{See also} \texttt{dde-item}, \texttt{dde-poke}, \texttt{dde-request}

\texttt{define-dde-client} \hfill \textit{Macro}

\begin{description}
\item[Summary] Registers a client service.
\item[Package] \texttt{win32}
\item[Signature] \texttt{define-dde-client name &key service class => name}
\end{description}
The DDE client interface

Arguments

name  A symbol.
service  A string.
class  A subclass of dde-client-conversation.

Values

name  A symbol.

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.

\textbf{See also} \hspace{1em} \texttt{define-dde-client}
The DDE server interface

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 or Mac OS X platforms.

**dde-server-poke**

*Generic Function*

**Summary**

Called when a poke transaction is received.

**Package**

win32
The DDE server interface

Signature

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     server    A server object.
               topic     A topic object.
               item      A string.
               format    A keyword.

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 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 transac-
tions 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
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 283 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`
The DDE server interface

This chapter applies only to LispWorks for Windows

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.
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.
This chapter applies only to LispWorks for Windows

Example

```lisp
(define-dde-dispatch-topic topic1 :server demo-server)
(define-dde-server-function (item1 :topic topic1)
  :request
  ()
  \ldots\text{handle topic1.item1 request}\ldots)
```

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**

```
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.
This chapter applies only to LispWorks for Windows

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”.

```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.

```lisp
(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
```

---

1761
The DDE server interface

### Arguments

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A symbol.</td>
</tr>
<tr>
<td>transaction</td>
<td>A keyword.</td>
</tr>
<tr>
<td>server</td>
<td>A server object.</td>
</tr>
<tr>
<td>topic-class</td>
<td>A topic class.</td>
</tr>
<tr>
<td>topic</td>
<td>A symbol naming a dispatch topic.</td>
</tr>
<tr>
<td>item</td>
<td>A string.</td>
</tr>
<tr>
<td>format</td>
<td>A keyword.</td>
</tr>
<tr>
<td>command</td>
<td>A string.</td>
</tr>
<tr>
<td>result-type</td>
<td>A data type.</td>
</tr>
<tr>
<td>advisep</td>
<td>A boolean.</td>
</tr>
<tr>
<td>var</td>
<td>A variable.</td>
</tr>
<tr>
<td>data-type</td>
<td>A data type.</td>
</tr>
<tr>
<td>type-spec</td>
<td>A data type.</td>
</tr>
<tr>
<td>form</td>
<td>A Lisp form.</td>
</tr>
</tbody>
</table>

### Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A symbol.</td>
</tr>
</tbody>
</table>

### 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
should be used. It can be a symbol which names a 
\textbf{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 conversation. 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 provided 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 specify the format understood. It defaults to \texttt{:text}. It can be specified 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 definition 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 transactions. They specify the arguments expected. \texttt{type-spec} should be one of \texttt{t, string, number, integer} or \texttt{float}. If not specified, \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**

<table>
<thead>
<tr>
<th>name</th>
<th>A DDE server class</th>
</tr>
</thead>
</table>

**Values**

<table>
<thead>
<tr>
<th>server</th>
<th>A server object</th>
</tr>
</thead>
</table>

**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
Dynamic library C functions

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 32-bit LispWorks on Microsoft Windows, Intel Macintosh, Linux, x86/x64 Solaris and FreeBSD, and 64-bit LispWorks on Windows, Intel Macintosh, Linux and x86/x64 Solaris.

### 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:
int InitLispWorks (int MilliTimeOut, void *BaseAddress, size_t ReserveSize)

Description

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 396)

`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 396.

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.
-2 LispWorks already started initialization, and \texttt{InitLispWorks} timed out before LispWorks finished mapping itself from the file.

-3 \texttt{InitLispWorks} started initialization and timed out after LispWorks mapped itself from the file, but before the initialization was complete.

-4 LispWorks already started initialization, and \texttt{InitLispWorks} timed out before 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:

-1000 Failure to start a thread to do the initialization.

-1401 The file seems to be corrupted.

-1402 Failure to map into memory.

-1403 Failure to read the LispWorks header from the file.

-1406 Bad base address.

Additionally, a value \texttt{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 \(-1001 - \texttt{value}\).

\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
Symbols
1 156
- 479
:< debugger command 15
$(dollar) variable 26
$$ variable 26
$$$ variable 26
:> debugger command 15
> SQL operator 312
?: listener command 7
 [...] syntax in Common SQL 311

Numerics
16-bit-string type 944
8-bit-string type 943

A
:a debugger command 20
abort restart 12
accepting-handle type 454
accepting-handle-collection function 455
accepting-handle-local-port function 456
accepting-handle-name function 456
accepting-handle-socket function 457
accepting-handle-user-info function 458
accept-tcp-connections-creating-async-io-states function 451
accessor-method-slot-definition generic function 214

accessors
code-coverage-data-create-time 749
code-coverage-data-name 749
code-coverage-file-stats-called 757
code-coverage-file-stats-counters-count 757
code-coverage-file-stats-counters-executed 757
code-coverage-file-stats-counters-hidden 757
code-coverage-file-stats-fully-covered 757
code-coverage-file-stats-hidden-covered 757
code-coverage-file-stats-lambdas-count 757
code-coverage-file-stats-not-called 757
code-coverage-file-stats-partially-covered 757
code-coverage-file-stats-source-file 757
dde-item 1739
dde-item* 1740
fdf-handle-directory-p 803
fdf-handle-directory-string 803
fdf-handle-last-access 803
fdf-handle-last-modify 803
fdf-handle-link-p 803
fdf-handle-size 803
fdf-handle-writable-p 803
field-access-exception-set-p 1092
Index

field-exception-class-name 1093
field-exception-field-name 1093
java-array-error-array 1117
java-array-error-caller 1117
java-array-indices-error-indices 1117
java-array-indices-error-rank 1117
java-bad-jobobject-caller 1119
java-bad-jobobject-object 1119
java-definition-error-class-name 1120
java-definition-error-name 1120
java-exception-exception-name 1121
java-exception-java-backtrace 1121
java-exception-string 1121
java-field-error-field-name 1120
java-field-error-static-field-name 1120
java-field-setting-error-class-name 1122
java-field-setting-error-class-name-for-setting 1122
java-field-setting-error-field-name 1122
java-field-setting-error-new-value 1122
java-instance-jobobject 185, 1165
java-method-error-args-num 1120
java-method-error-method-name 1120
java-method-exception-args 1125
java-method-exception-class-name 1125
java-method-exception-method-name 1125
java-method-exception-name 1125
lob-stream-lob-locator 1410
socket-stream-socket 536
sql-database-data-error 1475
sql-error-database-message 1475
sql-error-secondary-error-id 1475
storage-exhausted-gen-num 1668
storage-exhausted-size 1668
storage-exhausted-static 1668
storage-exhausted-type 1668
stream-read-timeout 536
stream-write-timeout 536
action lists 77
defining 78
definitions 78
undefining 78
*active-finders* variable 671
add-code-coverage-data function 719
adding actions to action lists 78
addMessage Java method 1198
add-method generic function 215
address space
  in 32-bit LispWorks 396, 398
  in 64-bit LispWorks 398, 415
add-special-free-action function 134, 721
add-sql-stream function 319, 1361
add-symbol-profiler function 139, 722
adjust-array function 230
advice
  after 51
  around 51
  before 51
example of use 55
facility 49
for macros 53
for methods 53
main chapter 49, 59
removing 52
after advice 51
:after keyword 38
:all debugger command 18
:all keyword 263
all SQL operator 312
:allocation keyword 43
allocation of stacks 134, 1550
allocation-in-gen-num macro 119, 128, 723
allowing-block-interrupts macro 245, 1205
analysing-special-variables-
usage macro 724
Android interface
Java side 1185
overview 189
Android runtimes
creating 190
example 193
android-funcall-in-main-
thread function 728
android-funcall-in-main-
thread-list function 728
android-get-current-activity
function 729
android-main-thread-p
function 731
ANSI
Common Lisp 216, 302
SQL mode 288, 322
ANSI Common Lisp Standard
menu command 561
ANSI_QUOTES
SQL mode 322
any SQL operator 312
any-capi-window-displayed-p
function 732
any-other-process-non-inter-
nal-server-p function 1207
append macro 945
append-file function 944
apply-in-pane-process
function 231
apply-in-wait-state-collec-
tion-process function 458
apply-with-allocation-in-
gen-num function 129, 1529
approaching-memory-limit
condition 1530
apropos function 561
apropos-list function 562
argument list 998
arguments
command line 392
lisp function 998
arguments for traced functions 38
around advice 51
array-dimension-limit
constant 417
array-single-thread-p
function 732
array-total-size-limit
constant 417
array-weak-p function 733
ASCII 382, 385
:ascii external format 382
ASDF 260, 265
ASDF2 260, 265
async-io-state structure type 459
async-io-state-abort
function 460
async-io-state-abort-and-
close function 462
async-io-state-address
function 463
async-io-state-buffered-
data-length function 464
async-io-state-collection
function 460
async-io-state-discard
function 464
async-io-state-finish
function 465
async-io-state-get-buffered-
data function 466
async-io-state-max-read
function 460
async-io-state-name function 460
async-io-state-object
function 460
async-io-state-old-length
function 460
async-io-state-peer-address
function 467
async-io-state-read-buffer
function 468
async-io-state-read-status
function 460
async-io-state-read-timeout
function 460
async-io-state-receive-mes-
sage function 473
async-io-state-send-message
function 475
async-io-state-send-message-
to-address function 477
async-io-state-user-info
function 363, 460
async-io-state-write-buffer
function 479
async-io-state-write-status
function 460
async-io-state-write-timeout
function 460
at-location macro 672
Index

atomic-decf macro 1531
atomic-exchange macro 1532
atomic-fixnum-decf macro 1532
atomic-incf macro 1531
atomicity and thread-safety in CAPI 231
atomic-push macro 1534
atomic-pop macro 1533
attribute-type function 305, 1362
augmented-string function 376, 563
augmented-string-p function 949
*autoload-asdf-integration* variable 945
base slot 306
base-char type 374, 946
base-character type 946
base-character-p function 946
base-char-code-limit constant 947
base-char-p function 947
base-char-ref function 1623
base-string type 376, 563
base-string-p function 948

:base-table class option 1387
before advice 51
:before keyword 37
binary files 83
*binary-file-type* variable 571, 1536
*binary-file-types* variable 571, 1536
:bindings keyword 655
binds-who function 738
block-promotion macro 739
.bmp external format 382, 707
.bmp-char type 374, 948
.bmp-char-p function 949
.bmp-native external format 384, 707
.bmp-reversed external format 384, 707
.bmp-string type 376, 950
.bmp-string-p function 951
BOM 386, 388
Bordeaux threads
APIs needed for 1282
:bg debugger command 15
:break keyword 39
break-new-instances-on-access function 423
break-on-access function 424
:break-on-exit keyword 39
break-on-unresolved-functions function 937
browser 951
*browser-location* variable 951
browsing documentation 951
buffered-stream class 1495
:bug-form listener command 7
BugFormLogsList Java class 1202
BugFormViewer Java class 1203
building-universal-intermediate-p function 740
byte order
big-endian architectures 932
little-endian architectures 932
Byte Order Mark 386, 388

C
: c debugger command 20
C functions
dlopen 159
dlsym 159
GetProcAddress 159
InitLispWorks 162, 397, 1767
LispWorksDlsym 1770
Index

LispWorksState 162, 1771
LoadLibrary 159
QuitLispWorks 163, 1773
SimpleInitLispWorks 1772
cache-table-queries function 301, 1364
*cache-table-queries-default* variable 1365
call Unix functions from Lisp 400
callDoubleA Java method 1180
callDoubleV Java method 1180
calling AppleScript 1538
callIntA Java method 1180
callIntV Java method 1180
call-java-method function 1065
call-java-method-error condition 1066
call-next-advice function 52, 58, 952
callObjectA Java method 1180
callObjectV Java method 1180
calls-who function 741
call-system function 1537
call-system-showing-output function 1540
callVoidA Java method 1180
callVoidV Java method 1180
class options parsing of 433
:base-table 1387
:extra-initargs 425, 427, 579
:optimize-slot-access 214, 217, 442, 579
classes buffered-stream 1495
dde-system-topic 1757
dde-topic 1757
eql-specializer 214
callable-standard-object 432
fundamental-binary-input-stream 1497
fundamental-binary-output-stream 1498
fundamental-binary-stream 1498
fundamental-character-input-stream 1499
fundamental-character-output-stream 1500
fundamental-character-stream 1501
fundamental-input-stream 1501
fundamental-output-stream 1502
fundamental-stream 1502
lob-stream 327, 331, 336, 1409
method-combination 215
socket-error 535
socket-stream 351, 356, 535
ssl-closed 361, 545
ssl-condition 361, 545
ssl-error 361, 546
ssl-failure 361, 546
ssl-x509-lookup 361, 548
standard-accessor-method 214
standard-db-object 305, 1484
standard-java-object 1165
standard-reader-method 214
standard-writer-method 214

check-java-field function 1156
check-lisp-calls-initialized function 1069
checkLispSymbol Java method 1183
*check-network-server*
variable 1544
choose-unicode-string-hash-function function 954

call-who function 1364
*cache-table-queries-default* variable 1365
call Unix functions from Lisp 400
callDoubleA Java method 1180
callDoubleV Java method 1180
calling AppleScript 1538
callIntA Java method 1180
callIntV Java method 1180
call-java-method function 1065
call-java-method-error condition 1066
call-next-advice function 52, 58, 952
callObjectA Java method 1180
callObjectV Java method 1180
calls-who function 741
call-system function 1537
call-system-showing-output function 1540
callVoidA Java method 1180
callVoidV Java method 1180
class options parsing of 433
:base-table 1387
:extra-initargs 425, 427, 579
:optimize-slot-access 214, 217, 442, 579
classes buffered-stream 1495
dde-system-topic 1757
dde-topic 1757
eql-specializer 214
callable-standard-object 432
fundamental-binary-input-stream 1497
fundamental-binary-output-stream 1498
fundamental-binary-stream 1498
fundamental-character-input-stream 1499
fundamental-character-output-stream 1500
fundamental-character-stream 1501
fundamental-input-stream 1501
fundamental-output-stream 1502
fundamental-stream 1502
lob-stream 327, 331, 336, 1409
method-combination 215
socket-error 535
socket-stream 351, 356, 535
ssl-closed 361, 545
ssl-condition 361, 545
ssl-error 361, 546
ssl-failure 361, 546
ssl-x509-lookup 361, 548
standard-accessor-method 214
standard-db-object 305, 1484
standard-java-object 1165
standard-reader-method 214
standard-writer-method 214
Index

storage-exhausted 1668
wait-state-collection 557
class-extra-initargs generic function 425
clean-down function 118, 133, 744
clean-generation-0 function 128, 746
clearBugFormLogs Java method 1196
clear-code-coverage function 747
CLOS Metaobject Protocol
  menu command 213
close generic function 563
close-accepting-handle function 482
close-async-io-state function 483
close-registry-key function 406, 1705
close-serial-port function 1353
close-wait-state-collection function 484
:clos-initarg-checking delivery keyword 439
Cocoa application 1241
  Cocoa application bundle
    saving 147
  Cocoa event loop 1241
Code Coverage
  Editor command 762
  File Editor command 762
code signing
  in saved image 873
  code-coverage-data type 748
code-coverage-data-create-time accessor 749
code-coverage-data-generate-coloring-html function 750
code-coverage-data-generate-statistics function 756
code-coverage-data-name accessor 749
code-coverage-data-stats type 757
code-coverage-data-stats-called accessor 757
code-coverage-file-stats-counters-count accessor 757
code-coverage-file-stats-counters-executed accessor 757
code-coverage-file-stats-counters-hidden accessor 757
code-coverage-file-stats-fully-covered accessor 757
code-coverage-file-stats-hidden-covered accessor 757
code-coverage-file-stats-lambdas-count accessor 757
code-coverage-file-stats-not-called accessor 757
code-coverage-file-stats-partially-covered accessor 757
code-coverage-file-stats-source-file accessor 757
code-coverage-set-editor-colors function 760
code-coverage-set-editor-default-data function 761
code-coverage-set-html-background-colors function 762
code-page external format 381
coerce function 564
coerce-to-gesture-spec function 1544
*coff-loading-verbose* 942
collect-generation-2 function 126, 128, 763
collect-highest-generation function 128, 764
collect-registry-subkeys function 406, 1706
collect-registry-values function 406, 1707
command line 392
command line arguments 1604
  -build 392
  -display 393
  -env 393
  -environment 393
  -eval 393
  -IIOPhost 393
  -IIOPnumeric 393
  -init 394
  -load 394
-lw-no-redirection 394
-multiprocessing 394
-no-restart-function 394
-ORBport 394
--relocate-image 395
--reserve-size 395
-siteinit 395
command line processing 1605
commands
   listener 1552
top level 1552
commit function 300, 302, 329, 1365
Common Lisp
   systems. See system
Common SQL
   [...] syntax 311
case of names 321
database connection 292
database encoding 322
date fields 317, 326
encoding 293
errors 320
Functional DDL 305
Functional DML 298
functional interface 297
I/O recording 319
initialization 288
iteration 310
main chapter 285
Object Oriented DDL 306
Object Oriented DML 309
object-oriented interface 305
ODBC compliance 287
programmatic interface 315
result types 298, 299, 325
supported databases 287
symbolic syntax 310
transaction handling 293, 301, 324
utilities 317
Common SQL errors
   sql-connection-error 320
   sql-database-data-error 320
   sql-database-error 320
   sql-fatal-error 320
   sql-temporary-error 320
   sql-timeout-error 320
   sql-user-error 320
   compare-and-swap macro 1546
   compilation-speed 87
   compile function 565
   compile-file function 567
   compile-file-if-needed function 766
   compiler
      comparison with interpreter 83
      control 86
      levels of safety 87
      main chapter 83
      optimization of 86–90
      workings of 85
   compiler explanations 93, 573
   compiler help 93, 573
   *compiler-break-on-error*
      variable 765
   compile-system function 260, 954
   compiling
      arbitrary forms 85
      functions 84
      source files 85
   compute-applicable-methods-
      using-classes generic function 215
   compute-class-potential-init-
      targs generic function 426
   compute-discriminating-func-
      tion generic function 215, 428
   compute-effective-method-
      function-from-classes
      generic function 428
   concatenate function 572
   concatenate-system function 956
   Condition variables 241
   Conditionalization
      LispWorks architectures 596
      LispWorks implementations 593
      LispWorks versions 594
   Conditionalization for LispWorks
      versions 594
   Conditionalization for the LispWorks
      architectures 596
   Conditionalization for the LispWorks
      implementations 593
   conditions
      approaching-memory-
         limit 1530
      call-java-method-error 1066
      create-java-object-
         error 1073
      external-format-error 715
      fasl-error 802
      field-access-exception 1092
      field-exception 1093
      file-encoding-resolution-
         error 1563
java-array-error 1116
java-array-indices-error 1117
java-array-simple-error 1118
java-bad-jobct 1119
java-class-error 1120
java-definition-error 1120
java-exception 1121
java-field-error 1120
java-field-setting-error 1122
java-ld-exception 1123
java-instance-without-job-ject-error 1123
java-interface-error 1124
java-low-level-exception 1125
java-method-error 1120
java-method-exception 1125
java-normal-exception 1126
java-not-a-java-object-error 1127
java-not-an-array-error 1127
java-out-of-bounds-error 1130
java-serious-exception 1131
java-storing-wrong-type-error 1130
sql-connection-error 1474
sql-database-data-error 1474
sql-database-error 1475
sql-fatal-error 1477
sql-temporary-error 1483
sql-timeout-error 1483
sql-user-error 1483
condition-variable-broadcast function 241, 1221
condition-variable-signal function 241, 1221
condition-variable-wait function 241, 1222
condition-variable-wait-count function 241, 1224
configuring the printer 157
connect function 288, 292, 293, 321, 1366
connected-databases function 292, 1374
*connect-if-exists* variable 1373
connecting to a database
MySQL 293
ODBC 293
Oracle 292

PostgreSQL 296
connect-to-named-pipe function 1678
connect-to-tcp-server function 495
console 869
console application 869
constants
array-dimension-limit 417
array-total-size-limit 417
base-char-code-limit 947
most-positive-fixnum 417
continue restart 12
copy-code-coverage-data function 767
copy-current-code-coverage function 767
copy-file function 958
copy-preferences-from-older-version function 1547
copy-standard-object function 431
copy-to-weak-simple-vector function 135, 769
count-gen-num-allocation function 132, 1548
Counting semaphores 243
create-and-run-wait-state-collection function 484
create-async-io-state function 487
create-async-io-state-and-connected-tcp-socket function 489
create-async-io-state-and-connected-udp-socket function 491
create-async-io-state-and-udp-socket function 493
create-index function 305, 1375
create-instance-from-jobct function 1069
create-instance-jobct function 1070
create-instance-jobct-list function 1070
create-java-jobct function 1072
create-java-object-error condition 1073
createLispProxy Java method 1182
create-macos-application-bundle function 770
<table>
<thead>
<tr>
<th>create-registry-key function</th>
<th>406, 1709</th>
</tr>
</thead>
<tbody>
<tr>
<td>create-table function</td>
<td>305, 1376</td>
</tr>
<tr>
<td>create-temp-file function</td>
<td>405, 773</td>
</tr>
<tr>
<td>create-universal-binary function</td>
<td>775</td>
</tr>
<tr>
<td>create-view function</td>
<td>305, 1377</td>
</tr>
<tr>
<td>create-view-from-class function</td>
<td>307, 1378</td>
</tr>
<tr>
<td>creation of process</td>
<td>226</td>
</tr>
<tr>
<td>cross-referencing</td>
<td>738, 741, 905, 923, 924</td>
</tr>
<tr>
<td>:ctx-configure-callback initarg</td>
<td>536</td>
</tr>
<tr>
<td>current frame</td>
<td>15</td>
</tr>
<tr>
<td>current process</td>
<td>226</td>
</tr>
<tr>
<td>current-pathname function</td>
<td>959</td>
</tr>
<tr>
<td><em>current-process</em> variable</td>
<td>226, 1224</td>
</tr>
<tr>
<td>current-process-block-interrupts function</td>
<td>245, 1224</td>
</tr>
<tr>
<td>current-process-in-cleanup-p function</td>
<td>1226</td>
</tr>
<tr>
<td>current-process-kill function</td>
<td>1226</td>
</tr>
<tr>
<td>current-process-pause function</td>
<td>1227</td>
</tr>
<tr>
<td>current-process-send function</td>
<td>1229</td>
</tr>
<tr>
<td>current-process-set-terminate-method function</td>
<td>1230</td>
</tr>
<tr>
<td>current-process-unblock-interrupts function</td>
<td>245, 1231</td>
</tr>
<tr>
<td>current-stack-length function</td>
<td>775</td>
</tr>
<tr>
<td>customization main chapter</td>
<td>146</td>
</tr>
<tr>
<td>of editor</td>
<td>153</td>
</tr>
<tr>
<td>:cv inspector command</td>
<td>27</td>
</tr>
<tr>
<td>dde-advertise-start function</td>
<td>278, 1723</td>
</tr>
<tr>
<td>dde-advertise-start* function</td>
<td>278, 1726</td>
</tr>
<tr>
<td>dde-advertise-stop function</td>
<td>279, 1727</td>
</tr>
<tr>
<td>dde-advertise-stop* function</td>
<td>279, 1729</td>
</tr>
<tr>
<td>dde-client-advertise-data generic function</td>
<td>1730</td>
</tr>
<tr>
<td>dde-connect function</td>
<td>277, 1731</td>
</tr>
<tr>
<td>dde-disconnect function</td>
<td>277, 1732</td>
</tr>
<tr>
<td>dde-execute function</td>
<td>1732</td>
</tr>
<tr>
<td>dde-execute* function</td>
<td>1733</td>
</tr>
<tr>
<td>dde-execute-command function</td>
<td>280, 1734</td>
</tr>
<tr>
<td>dde-execute-command* function</td>
<td>280, 1735</td>
</tr>
<tr>
<td>dde-execute-string function</td>
<td>280, 1737</td>
</tr>
<tr>
<td>dde-execute-string* function</td>
<td>280, 1737</td>
</tr>
<tr>
<td>dde-item accessor</td>
<td>1739</td>
</tr>
<tr>
<td>dde-item* accessor</td>
<td>1740</td>
</tr>
<tr>
<td>dde-item function</td>
<td>280</td>
</tr>
<tr>
<td>dde-poke function</td>
<td>279, 1742</td>
</tr>
<tr>
<td>dde-poke* function</td>
<td>1744</td>
</tr>
<tr>
<td>dde-request function</td>
<td>279, 1746</td>
</tr>
<tr>
<td>dde-request* function</td>
<td>1748</td>
</tr>
<tr>
<td>dde-server-poke generic function</td>
<td>281, 1753</td>
</tr>
<tr>
<td>dde-server-request generic function</td>
<td>281, 1754</td>
</tr>
<tr>
<td>dde-server-topic generic function</td>
<td>1755</td>
</tr>
<tr>
<td>dde-server-topics generic function</td>
<td>282, 1756</td>
</tr>
<tr>
<td>dde-system-topic class</td>
<td>1757</td>
</tr>
<tr>
<td>dde-topic class</td>
<td>1757</td>
</tr>
<tr>
<td>dde-topic-items generic function</td>
<td>1758</td>
</tr>
<tr>
<td>DDL</td>
<td>305, 306</td>
</tr>
<tr>
<td>debugger</td>
<td>87</td>
</tr>
<tr>
<td>break gesture</td>
<td>10</td>
</tr>
<tr>
<td>commands</td>
<td>13</td>
</tr>
<tr>
<td>commands not recognized</td>
<td>23</td>
</tr>
<tr>
<td>control variables</td>
<td>23</td>
</tr>
<tr>
<td>invoking from the tracer</td>
<td>39</td>
</tr>
<tr>
<td>keyboard interrupt</td>
<td>10</td>
</tr>
<tr>
<td>main chapter</td>
<td>9</td>
</tr>
<tr>
<td>troubleshooting</td>
<td>23</td>
</tr>
<tr>
<td>debugger commands</td>
<td>15</td>
</tr>
<tr>
<td>:&lt;</td>
<td>15</td>
</tr>
<tr>
<td>:&gt;</td>
<td>15</td>
</tr>
</tbody>
</table>
Index

*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 1549

*debug-io* variable 23, 623

debug-other-process function 1232

*debug-print-length* variable 23, 647

*debug-print-level* variable 24, 648

declare macro 92, 572
declaration alias 573
:oexplain* 573
:invisible-frame 573
:lambda-list 573
:special-dynamic* 95, 573
:special-fast-access* 95, 573
:special-global* 95, 573
:values 573
declaration-information function 776

declare :explain 93, 573
declare special form 86, 91, 573
decode-external-string function 714

decode-to-db-standard-date function 1380
decode-to-db-standard-times-tamp function 1380
def macro 674
defadvice macro 53, 58, 960
default directory 824
default file directory 824

*default-action-list-sort-time* variable 964, 79

*default-character-element-type* parameter 376, 378, 379, 380, 387, 964

default-constructor-arguments generic function 1073

*default-database* variable 1382

*default-database-type* variable 1382

default-eol-style function 1549

*default-libraries* variable 941
default-name-constructor function 1074

*default-package-use-list* variable 777

default-pathname keyword 261

*default-process-priority* variable 1233

*default-profiler-collapse* variable 778

*default-profiler-cutoff* variable 778

*default-profiler-limit* variable 779

*default-profiler-sort* variable 779

*default-stack-group-list-length* variable 134, 1550

*default-update-objects-max-length* variable 1383
defclass macro 578
defglobal-parameter macro 780
defglobal-variable macro 780
define-action macro 78, 965
define-action-list macro 78, 967
define-action-list macro 78, 967
define-atomic-modify-macro macro 1551
define-dde-client function 279
define-dde-client macro 1749
define-dde-dispatch-topic macro 283, 1759
define-dde-server function macro 281, 1760
define-dde-server-function macro 281, 1761
define-dspec-alias macro 675
define-dspec-class macro 676
define-field-accessor macro 1075
define-foreign-callable macro 252
define-form-parser macro 72, 679

1784
Index

define-java-caller macro 1073
define-java-callers macro 1079
define-java-constructor macro 1076
define-lisp-proxy macro 1081
define-top-loop-command macro 1552
definition specs 46
defpackage macro 582
defparameter macro 780
defparser error handling with 270
defparser macro 1349
functions defined by 269
defrule compiling a rule dynamically 85
defstruct macro 157
*defstruct-generates-print-object-method* variable 157
defsystem macro 259–265, 969
elements of use 264
*defsystem-verbose* variable 975
deftransform macro 643
defvar macro 781
def-view-class macro 286, 306, 1383
delete-advice macro 52, 58, 781
delete-directory function 976
delete-duplicates function 98
delete-instance-records generic function 309, 1390, 1404
delete-records function 300, 301, 1391
delete-registry-key function 406, 1710
delete-sql-stream function 319, 1392
deliver function 159, 392, 397, 976
deliverable
filename 392, 1009
pathname 392, 1009
delivered-image-p function 782
delivering a DLL 159
delivering a dynamic library 159
deliver-to-android-project function 783
delivery keywords
:closer-initarg-checking 439
:dll-added-files 184
:multithreading 228
:quit-when-no-windows 1033
:startup-bitmap-file 1680
describe function 25, 584
*describe-length* variable 977, 27
*describe-level* variable 978
*describe-level* special variable 26
describe-object generic function 26
*describe-print-length* variable 980, 26
*describe-print-level* variable 980, 26
destroy-prepared-statement function 1393
destroy-ssl function 359, 498
destroy-ssl-ctx function 359, 498
destructive-add-code-coverage-data function 719
destructive-reverse-subtract-code-coverage-data function 719
detach-sql function 358, 498
detect-eol-style function 1554
detect-japanese-encoding-in-file function 1556
detect-unicode-bom function 1556
detect-utf32-bom function 1556
detect-utf8-bom function 1556
diagnostic utilities
for action lists 79
:direction initarg 535, 1409, 1495
directory function 584
*directory-link-translation* special variable 585
*directory-link-translation* variable 1558
disable-sql-reader-syntax function 317, 1393
*disable-trace* variable 788
disassemble function 589
discard-source-info function 683
disconnect function 292, 1394
dismiss-splash-screen function 1680
DLL 159
filename 392, 1009
pathname 392, 1009
:dll-added-files delivery keyword 184
dll-quit function 163, 980
dlopen C function 159
dlsym C function 159
Index

:dm inspector command 27
DML 298, 309
DNS 502
documentation generic function 589
$(dollar) variable 26
$$ variable 26
$$$ variable 26
domain 502
do-nothing function 984
:dont-know keyword 17
do-profiling function 139, 788
do-query macro 302, 328, 1395
do-rand-seed function 500
DOS command
call-system 1537
call-system-showing-output 1540
open-pipe 1625
DOS window controlling, in call-system 1538, 1539
dotted-list-length function 983
dotted-list-p function 984
double-float type 590
:dr inspector command 27
drop-index function 305, 1396
drop-table function 305, 1397
drop-view function 305, 1397
drop-view-from-class function 307, 1398
dspec-class function 683
*dspec-classes* variable 684
dspec-defined-p function 684
dspec-definition-locations function 685
dspec-equal function 686
dspec-name function 686
dspec-primary-name function 687
dspec-progenitor function 688
dspecs aggregate 66 canonical 60 displaying definitions 71 examples 59 finding definitions 71 grouping definitions 66 new defining forms 64 parts 66 recording definitions 69
dspec-subclass-p function 688
dspec-undefiner function 689
dump-form function 930
dump-forms-to-file function 930
dylib 159 dynamic libraries 159, 396 dynamic library 159 memory clash 162 relocation 162
E
:ed debugger command 18
editor customizing 153
ditor source code 154
editor-color-code-coverage function 791
ef-spec 381
:ef-spec initarg 1563
:element-type initarg 535, 1496
Emacs 3
enable-sql-reader-syntax function 298, 317, 1399
encode-db-standard-date function 1381
encode-db-standard-timestamp function 1381
encode-lisp-string function 715
encoding changing default for files 386
enlarge-generation function 126, 128, 794
enlarge-static function 795
ensure-hash-entry function 796
ensure-is-jobject function 1102
ensure-lisp-classes-from-tree function 1089
ensure-loads-after-loads function 1558
ensure-memory-after-store function 1559
ensure-objc-initialized function 209
ensure-process-cleanup function 1233
ensure-ssl function 359, 500
ensure-stores-after-memory function 1560
ensure-stores-after-stores function 1560
ensure-supers-contain-java.lang.object function 1091
*enter-debugger-directly* variable 985
Index

:entrycond keyword 40
enum-registry-value function 406, 1711
environment access API 735, 777, 813, 923
environment-variable function 985
eql-specializer class 214
eql-specializer-object function 214
erro-value function 987
:error debugger command 18
error handlers
  in applications 654
error handling
  in parser generator 270
error output 1614
*error-output* variable 623
errors in Common SQL 320
error-situation-forms macro 797
EUC-JP 382
:eval-after keyword 39
:eval-before keyword 38
evaluating
  forms during tracing 37–39
example-compile-file function 988
example-edit-file function 989
example-file function 987
example-load-binary-file function 990
except SQL operator 312
except handlers
  in applications 654
except handling
  for action lists 78
exceptions
  handling 654
executable 392
  filename 392, 1009
  pathname 392, 1009
executable-log-file function 649
execute-actions macro 990
execute-command function 304, 1399
execute-with-interface function 231
execution functions 77
execution profiling 137
execution stack
  examining 12
:exitcond keyword 40
expand-generation-1 function 128,
  798
extend-current-stack function 799
extended-char type 992
extended-character type 992
extended-character-p function 992
extended-char-p function 993
*extended-spaces* variable 1060, 1561
extended-time macro 131, 133, 143, 800
external format
  changing default for files 386
external format specification 381
external formats 381
  :ascii 382
  :bmp 382, 707
  :bmp-native 384, 707
  :bmp-reversed 384, 707
code-page 381
  :euc-jp 382
  :gbk 383
  :jis 382
  :koi-8 383
  :latin-1 381
  :latin-1-safe 381
  :latin-1-terminal 381
  :macos-roman 382
  :sjis 382
  :unicode 382
  :utf-16 382
  :utf-16be 384, 710
  :utf-16le 384, 710
  :utf-16-native 384, 710
  :utf-16-reversed 384, 710
  :utf-32 382
  :utf-8 382
  :windows-cp936 382
external programs
  calling from Lisp 400
external-format-error condition 715
external-format-foreign-type function 716
*external-formats* variable 993
external-format-type function 716
:extra-initargs class option 425, 427, 579
Index

F
false function 995
fasl (fast load)
description 83
fasl-error condition 802
fast-directory-files function 803
FDDL 305
fdf-handle-directory-p accessor 803
fdf-handle-directory-string accessor 803
fdf-handle-last-access accessor 803
fdf-handle-last-modify accessor 803
fdf-handle-link-p accessor 803
fdf-handle-size accessor 803
fdf-handle-writable-p accessor 803
FDML 309
*features* variable 591
field-access-exception condition 1092
field-access-exception-set-p accessor 1092
field-exception condition 1093
field-exception-class-name accessor 1093
field-exception-field-name accessor 1093
file descriptor, of socket-stream 536
file descriptor, on UNIX 1272, 1573
file-directory-p function 995
*file-encoding-detection-algorithm* variable 386, 1562
file-encoding-resolution-error condition 1563
*file-eol-style-detection-algorithm* variable 386, 1564
filename of deliverable 392, 1009
filename of DLL 392, 1009
filename of dynamic library 1009
filename of executable 392, 1009
filename of lisp image 392, 1009
*filename-pattern-encoding-matches* variable 1564
files
load-on-demand 156
file-stat-blocks function 1574
file-stat-device function 1574
file-stat-device-type function 1575
file-stat-group-id function 1574
file-stat-inode function 1574
file-stat-last-access function 1574
file-stat-last-change function 1574
file-stat-last-modify function 1574
file-stat-links function 1574
file-stat-mode function 1574
file-stat-owner-id function 1574
file-stat-size function 1574
file-string function 805
file-writable-p function 806
(setf fill-pointer) function 230
filter-code-coverage-data function 807
Find Source
menu command 679
find-database function 292, 1400
find-dspec-locations function 689
find-encoding-option function 1564
find-external-char function 717
find-filename-pattern-encoding-match function 1563
find-java-class function 1094
find-name-locations function 690
find-object-size function 127, 808
find-process-from-name function 1235
find-regexp-in-string function 996
find-throw-tag function 809
finish-heavy-allocation function 810
fixnum type 415
fixnum-safety 87
flag-not-special-free-action function 134, 810
flag-special-free-action function 134, 811
FLI types
lpcstr 1698
lpcstr 1699
lpcwstr 1700
lpstr 1698
lptstr 1699
lpwstr 1700
p-oci-env 332, 1457
Index

p-oci-file 332, 1458
p-oci-lob-locator 332, 1458
p-oci-lob-or-file 1458
p-oci-svc-ctx 332, 1458
ssl-cipher-pointer 353, 544
ssl-cipher-pointer-stack 544
ssl-ctx-pointer 353, 546
ssl-pointer 353, 547
str 1698
tstr 1699
wstr 1700
float 87
float calculations, optimizing 94
foreign callbacks 252, 253
foreign-slot-value function 98
foreign-symbol-address function 938
format-to-java-host function 1095
forms evaluating when tracing 37–39
frame, examining 12
:free-lob-locator-on-close initarg 1409
:func debugger command 19
funcallable-standard-class class 215
funcallable-standard-instance-access function 214
funcallable-standard-object class 216, 432
funcall-async function 1236
funcall-async-list function 1236
function, altering with advice 49
Functional DDL 305
Functional DML 298
functional interface in Common SQL 297
function-information function 812
function-lambda-list function 998
functions accepting-handle-collection 455
accepting-handle-local-port 456
accepting-handle-name 456
accepting-handle-socket 457
accepting-handle-user-info 458
accept-tcp-connections-creating-async-io-states 451
add-code-coverage-data 719
add-special-free-action 134, 721
add-sql-stream 319, 1361
add-symbol-profile 139, 722
android-funcall-in-main-thread 728
android-funcall-in-main-thread-list 728
android-get-current-activity 729
android-main-thread-p 731
any-capi-window-displayed-p 732
any-other-process-non-internal-server-p 1207
append-file 944
apply-in-wait-state-collection-process 458
apply-with-allocation-in-gen-num 129, 1529
apropos 561
apropos-list 562
array-single-thread-p 732
array-weak-p 733
async-io-state-abort 460
async-io-state-abort-and-close 462
async-io-state-address 463
async-io-state-buffered-data-length 464
async-io-state-collection 460
async-io-state-discard 464
async-io-state-finish 465
async-io-state-get-buffered-data 466
async-io-state-max-read 460
async-io-state-name 460
async-io-state-object 460
async-io-state-old-length 460
async-io-state-peer-address 467
async-io-state-read-buffer 468
async-io-state-read-status 460
async-io-state-read-timeout 460
async-io-state-receive-
message 473
async-io-state-send-message 475
async-io-state-send-message-to-address 477
async-io-state-user-info 363, 460
async-io-state-write-buffer 479
async-io-state-write-message 473
async-io-state-write-status 460
async-io-state-write-timeout 460
attach-ssl 351, 356, 480
attribute-type 305, 1362
augmented-string-p 1535
augment-environment 733
avoid-gc 128, 735
barrier-arriver-count 1208
barrier-change-count 1212
barrier-count 1212
barrier-disable 243, 1213
barrier-enable 243, 1214
barrier-name 1214
barrier-pass-through 1215
barrier-unblock 1216
barrier-wait 242, 1217
base-character-p 946
base-char-p 947
base-character-p 947
base-char-p 947
base-string-p 948
binds-who 738
bmp-char-p 949
bmp-string-p 951
break-new-instances-on-access 423
break-on-access 424
break-on-unresolved 937
building-universal-intermediate-p 740
cache-table-queries 301, 1364
call-java-method 1065
call-next-advice 52, 58, 952
calls-who 741
call-system 1537
call-system-showing-output 1540
call-wait-state-collection 482
canonicalize-dspec 673
canonicalize-sid-string 1677
cdr-assoc 1543
change-directory 155, 743
change-process-priority 251, 1220
char-external-code 713
checked-read-java-field 1156
check-fragmentation 127, 128, 743
check-java-field 1156
check-lisp-calls-initialized 1069
choose-unicode-string-hash-function 954
clean-down 118, 133, 744
clean-generation-0 128, 746
clear-code-coverage 747
close-accepting-handle 482
close-asynco-state 483
close-registry-key 406, 1705
close-serial-port 1353
close-wait-state-collection 484
code-coverage-data-generate-coloring-html 750
code-coverage-data-generate-statistics 756
code-coverage-set-editor-colors 760
code-coverage-set-editor-default-data 761
code-coverage-set-html-background-colors 762
coerce 564
coerce-to-gesture-spec 1544
collect-generation-2 126, 128, 763
collect-highest-generation 128, 764
collect-registry-subkeys 406, 1706
collect-registry-values 406, 1707
commit 300, 302, 329, 1365
compile 565
compile-file 567
compile-file-if-needed 766
compile-system 260, 954
concatenate 572
concatenate-system 956
condition-variable-broadcast 241, 1221
condition-variable-signal 241, 1221
Index

condition-variable-wait 241, 1222
condition-variable-wait-count 241, 1224
connect 292, 293, 321, 1366
connected-databases 292, 1374
connect-to-named-pipe 1678
connect-to-tcp-server 495
copy-code-coverage-data 767
copy-current-code-coverage 767
copy-file 958
copy-preferences-from-older-version 1547
copy-standard-object 431
copy-to-weak-simple-vector 135, 769
count-gen-num-allocation 132, 1548
create-and-run-wait-state-collection 484
create-async-io-state 487
create-async-io-state-and-connected-tcp-socket 489
create-async-io-state-and-connected-udp-socket 491
create-async-io-state-and-udp-socket 493
create-index 305, 1375
create-instance-from-jobject 1069
create-instance-jobject 1070
create-instance-jobject-list 1070
create-java-object 1072
create-macos-application-bundle 770
create-registry-key 406, 1709
create-table 305, 1376
create-temp-file 405, 773
create-universal-binary 775
create-view 305, 1377
create-view-from-class 307, 1378
current-pathname 959
current-process-block-interrupts 245, 1224
current-process-in-cleanup-p 1226
current-process-kill 1226
current-process-pause 1227
current-process-send 1229
current-process-set-terminate-method 1230
current-process-unblock-interrupts 245, 1231
current-stack-length 775
database-name 292, 1379
dde-advice-start 1723
dde-advice-start* 1726
dde-advice-stop 1727
dde-advice-stop* 1729
dde-connect 1731
dde-disconnect 1732
dde-execute 1732
dde-execute* 1733
dde-execute-command 1734
dde-execute-command* 1735
dde-execute-string 1737
dde-execute-string* 1737
dde-poke 1742
dde-poke* 1744
dde-request 1746
dde-request* 1748
debug-other-process 1232
declaration-information 776
decode-external-string 714
decode-to-db-standard-date 1380
decode-to-db-standard-timestamp 1380
default-eol-style 1549
default-name-constructor 1074
delete-directory 976
delete-duplicates 98
delete-records 300, 301, 1391
delete-registry-key 406, 1710
delete-sql-stream 319, 1392
deliver 159, 392, 397, 976
delivered-image-p 782
deliver-to-android-project 783
describe 25, 584
destroy-prepared-statement 1393
destroy-ssl 359, 498
destroy-ssl-ctx 359, 498
destructive-add-code-coverage-data 719
destructive-reverse-subtract-code-coverage-data 719
destructive-subtract-code-
Index

coverage-data 719
detach-ssl 358, 498
detect-ssl 1554
detect-japanese-encoding-in-file 1556
detect-unicode-bom 1556
detect-utf8-bom 1556
directory 584
disable-sql-reader-syntax 317, 1393
disassemble 589
discard-source-info 683
disconnect 292, 1394
dismiss-splash-screen 1680
dll-quit 163, 980
do-nothing 984
do-profiling 139, 788
do-rand-seed 300
dotted-list-length 983
dotted-list-p 984
drop-index 305, 1396
drop-table 305, 1397
drop-view 305, 1397
drop-view-from-class 307, 1398
dspec-class 683
dspec-defined-p 684
dspec-definition-locations 685
dspec-equal 686
dspec-name 686
dspec-primary-name 687
dspec-progenitor 688
dspec-subclass-p 688
dspec-undefined 689
dump-form 930
dump-forms-to-file 930
dump-forms-to-file 930
editor-color-code-coverage 791
enable-sql-reader-syntax 298, 317, 1399
encode-db-standard-date 1381
encode-db-standard-timestamp 1381
encode-lisp-string 715
enlarge-generation 126, 128, 794
enlarge-static 795
ensure-hash-entry 796
ensure-is-objc 1102
ensure-lisp-classes-from-tree 1089
ensure-loads-after-loads 1558
ensure-memory-after-store 1559
ensure-objc-initialized 209
ensure-process-cleanup 1233
ensure-ssl 359, 500
ensure-stores-after-memory 1560
ensure-stores-after-stores 1560
ensure-supers-contain-java.lang.object 1091
enum-registry-value 406, 1711
environment-variable 985
eql-specializer-object 214
errno-value 987
example-compile-file 988
example-edit-file 989
example-file 987
example-load-binary-file 990
executable-log-file 649
execute-command 304, 1399
expand-generation-1 128, 798
extend-current-stack 799
extended-character-p 992
extended-char-p 993
external-format-foreign-type 716
external-format-type 716
false 995
fast-directory-files 803
file-directory-p 995
file-stat-blocks 1574
file-stat-device 1574
file-stat-device-type 1575
file-stat-group-id 1574
file-stat-inode 1574
file-stat-last-access 1574
file-stat-last-change 1574
file-stat-last-modify 1574
file-stat-links 1574
file-stat-mode 1574
file-stat-owner-id 1574
file-stat-size 1574
file-string 805
file-writable-p 806
filter-code-coverage-data 807
find-database 292, 1400
find-dspec-locations 689
find-encoding-option 1564
find-external-char 717
Index

find-filename-pattern-encoding-match 1565
find-java-class 1094
find-name-locations 690
find-object-size 127, 808
find-process-from-name 1235
find-regexp-in-string 996
find-throw-tag 809
finish-heavy-allocation 810
flag-not-special-free-action 134, 810
flag-special-free-action 134, 811
foreign-slot-value 98
foreign-symbol-address 938
format-to-java-host 1095
funcallable-standard-instance-access 214
funcall-async 1236
funcall-async-list 1236
function-information 812
function-lambda-list 998
gc-generation 118, 120, 128, 132, 815
gc-if-needed 128, 818
generate-code-coverage 819
generate-java-class-definitions 1095
generation-number 126, 1568
gen-num-segments-fragmentation-state 132, 1566
genSYM 134
gesture-spec-data 1570
gesture-spec-modifiers 1571
gesture-spec-p 1571
gesture-spec-to-character 1572
get-code-coverage-delta 814
get-current-process 226, 227, 1238
get-default-generation 128, 822
get-default-local-ipv6-address 501
get-file-stat 1573
get-folder-path 405, 1575
get-foreign-symbol 939
get-form-parser 691
get-gc-parameters 128, 822
get-gc-timing 897
gethash-ensuring 825
get-host-entry 502
get-ip-default-zone-id 505
get-java-machine 1101
get-jobject 1102
get-primitive-array-region 1103
get-process 227, 1239
get-process-private-property 1240
get-serial-port-state 1354
get-service-entry 505
get-socket-address 507
get-socket-peer-address 508
get-superclass-and-interfaces-tree 1105
get-temp-directory 824
get-unix-error 1002
get-user-profile-directory 405, 1578
get-verification-mode 508
get-working-directory 824
guess-external-format 386, 1579
hardcopy-system 1007
immediatep 1580
initialize-database-type 1401
initialize-multiprocessing 227, 1240
init-java-interface 1108
insert-records 300, 328, 1402
inspect 26
int32* 1582
int32- 411, 1582
int32+ 411, 1582
int32/ 1582
int32/= 1585
int32<< 1586
int32<= 1585
int32< 1585
int32<< 1586
int32<= 1585
int32= 1585
int32> 1585
int32>= 1585
int32-> 1586
int32-< 1584
int32-<= 1584
int32-<> 1587
int32-aref 1587
int32-logand 1588
int32-logandcl 1588
int32-logandccl 1588
int32-logbitp 1588
int32-logeqv 1588

1793
Index

int32-logior 1588
int32-lognand 1588
int32-lognor 1588
int32-lognot 1588
int32-logorc1 1588
int32-logorc2 1588
int32-logtest 1588
int32-logxor 1588
int32-minusp 1590
int32-plusp 1590
int32-to-int64 1591
int32-to-integer 1592
int32-zerop 1590
int64* 1593
int64- 412, 1593
int64-= 1593
int64/= 1596
int64=< 1596
int64<= 1597
int64=> 1596
int64>= 1596
int64-> 1597
int64+l 1595
int64+- 1595
int64-aref 1598
int64-logand 1599
int64-logandc1 1599
int64-logandc2 1599
int64-logbitp 1599
int64-logeqv 1599
int64-logior 1599
int64-lognand 1599
int64-lognor 1599
int64-lognot 1599
int64-logorc1 1599
int64-logorc2 1599
int64-logtest 1599
int64-logxor 1599
int64-minusp 1601
int64-plusp 1601
int64-to-int32 1602
int64-zerop 1601
integer-to-int32 1603
integer-to-int64 1604
interactive-stream-p 597
intern-and-export-list 1113
intern-eql-specializer 214
ip-address-string 509
ipv6-address 510
ipv6-address-p 511
ipv6-address-scope-id 511
ipv6-address-string 512
jaref 1114
java-array-element-type 1115
java-array-length 1118
java-field-class-name-for-setting 1156
java-object-array-element-type 1128
java-objects-eq 1129
java-primitive-array-element-type 1130
java-type-to-lisp-array-element-type 1132
jobject-class-name 1135
jobject-ensure-global 1136
jobject-of-class-p 1137
jobject-p 1138
jobject-pretty-class-name 1139
jobject-string 1139
jobject-to-lisp 1140
jvref 1141
known-sid-integer-to-sid-string 1683
last-callback-on-thread 1242
lisp-array-to-primitive-array 1153
lisp-array-type-to-java-type 1132
lisp-image-name 392, 1009
lisp-java-instance-p 1143
lisp-name-to-foreign-name 940
lisp-to-jobject 1144
list-all-processes 227, 1243
list-attributes 305, 1406
list-attributetypes 305, 1405
list-classes 309, 1407
list-sql-streams 319, 1408
list-tables 305, 1408
load-all-patches 1012
load-code-coverage-data 767
load-data-file 830
load-code-coverage-data 598
load-system 1013
load-system 1013
lob-stream-lob-locator 331
locale-dspec-p 692
locale-file-encoding 1605
Index

locally-disable-sql-reader-syntax 317, 1410
locally-enable-sql-reader-syntax 317, 1411
lock-and-condition-variable-broadcast 1244
lock-and-condition-variable-signal 1245
lock-and-condition-variable-wait 1246
lock-locked-p 1248
lock-name 232, 1250
lock-owned-by-current-process-p 1248
lock-owner 232, 1251
lock-recursively-locked-p 1250
lock-recursive-p 1249
log-bug-form 652
logs-directory 653
long-namestring 392, 1685, 1697
long-site-name 391, 599
loop-processing-wait-state-collection 513
low-level-atomic-place-p 1606
mailbox-count 1252
mailbox-empty-p 1253
mailbox-not-empty-p 1253
mailbox-peek 1254
mailbox-read 1255
mailbox-reader-process 1256
mailbox-send 1257
mailbox-wait 1257
mailbox-wait-for-event 1259
make-array 135, 232, 601
make-barrier 242, 1261
make-condition-variable 1262
make-gesture-spec 1607
make-hash-table 135, 232, 602
make-java-array 1144
make-java-instance 1145
make-lisp-proxy 1146
make-lisp-proxy-with-overrides 1146
make-lock 1263
make-mailbox 1265
make-mt-random-state 1016
make-named-timer 1266
make-ring 834
make-semaphore 243, 1267
make-sequence 606
make-simple-int32-vector 1612
make-simple-int64-vector 1613
make-ssl-ctx 356, 357, 512
make-stderr-stream 1606
make-symmetric-action-list 1014
make-wait-state-collection 514
map 607
map-all-processes 1269
map-all-processes-backtrace 1270
map-environment 1616
map-java-object-array 1149
map-process-backtrace 1270
map-processes 1271
map-query 302, 328, 1414
map-ring 837
mark-and-sweep 838
marking-gc 132, 1617
memory-growth-margin 128, 1619
merge 608
merge-code-coverage-data 841
merge-ef-specs 1620
modify-hash 842
mt-random 1017
mt-random-state-p 1018
name-defined-dspecs 693
name-definition-locations 694
named-pipe-stream-name 1686
name-only-form-parser 695
normal-gc 128, 843
notice-fd 1272
object-address 1621
object-pointer 1622
octet-ref 1623
open 609
open-named-pipe-stream 1687
open-pipe 1624
open-registry-key 406, 1714
open-serial-port 1351
openssl-version 522
open-tcp-stream 351, 356, 514
open-tcp-stream-using-java 518
open-temp-file 405, 773
open-url 1629
ora-lob-append 335, 1418
ora-lob-assign 334, 1419
ora-lob-char-set-form 333, 1419
ora-lob-char-set-id 1420
ora-lob-close 335, 1421
ora-lob-copy 335, 1422
ora-lob-create-empty 328, 334, 1423
ora-lob-create-temporary 336, 1424
ora-lob-disable-buffering 336, 1425
ora-lob-element-type 333, 1426
ora-lob-enable-buffering 336, 1427
ora-lob-env-handle 331, 1428
ora-lob-erase 335, 1428
ora-lob-file-close 335, 1429
ora-lob-file-close-all 335, 1430
ora-lob-file-exists 1431
ora-lob-file-get-name 1432
ora-lob-file-open 335, 1433
ora-lob-file-set-name 335, 1434
ora-lob-flush-buffer 336, 1435
ora-lob-free 334, 1436
ora-lob-free-temporary 336, 1437
ora-lob-get-buffer 331, 336, 1437
ora-lob-get-chunk-size 334, 1440
ora-lob-get-length 334, 1440
ora-lob-internal-lob-p 333, 334, 1441
ora-lob-is-equal 334, 1442
ora-lob-is-open 334, 1442
ora-lob-is-temporary 334, 336, 1443
ora-lob-load-from-file 335, 1444
ora-lob-lob-locator 331, 1445
ora-lob-locator-is-init 334, 1446
ora-lob-open 335, 1446
ora-lob-read-buffer 333, 336, 1447
ora-lob-read-foreign-buffer 331, 333, 336, 1451
ora-lob-read-into-plain-file 336, 1450
ora-lob-svc-ctx-handle 331, 1452
ora-lob-trim 335, 1452
ora-lob-write-buffer 333, 336, 1453, 1456
ora-lob-write-foreign-buffer 331, 333, 336
ora-lob-write-from-plain-file 336, 1455
output-backtrace 654
package-flagged-p 1629
parse-float 845
parse-form-dspec 696
pathname-location 1019
pem-read 352, 524
pipe-exit-status 1630
pipe-kill-process 1631
pointer-from-address 1633
position-in-ring 846
position-in-ring-forward 846
precompile-regexp 1020
prepare-statement 1459
primitive-array-to-lisp-array 1153
print-action-lists 79, 1021
print-actions 79, 1020
print-pretty-gesture-spec 1634
print-profile-list 142, 848
print-query 299, 1461
process-alive-p 1272
process-all-events 1273
process-allow-scheduling 226, 1274
process-arrest-reasons 1274
process-break 1275
process-continue 1275
processes-count 227, 1316
process-exclusive-lock 1276
process-exclusive-unlock 1277
process-idle-time 1278
process-internal-server-p 1279
process-interrupt 245, 1280
process-interrupt-list 1281
Index

process-join 1282
process-kill 1283
process-lock 232, 1284
process-mailbox 1285
process-name 227, 1285
process-p 1286
process-plist 250, 1286
process-poke 1287
process-priority 251, 1289
process-private-property 1290
process-property 250, 1291
process-run-function 226, 1293
process-run-reasons 1296
process-send 1298
process-sharing-lock 1299
process-sharing-unlock 1300
process-stop 251, 1301
process-stopped 252
process-stopped-p 1302
process-terminate 1302
process-unlock 1304
process-unstop 251, 1305
process-wait 240, 1306
process-wait-for-event 1307
process-wait-function 1308
process-wait-local 1308
process-wait-local-with-periodic-checks 1311
process-wait-local-with-timeout 1313
process-wait-local-with-timeout-and-periodic-checks 1314
process-wait-with-timeout 236, 240, 1314
process-whostate 1316
proclaim 86, 91, 612
product-registry-path 402, 1636
profiler-tree-from-function 854
profiler-tree-to-function 855
ps 227, 1319
pushnew-to-process-private-property 250, 1317
pushnew-to-process-property 250, 1318
query 304, 327, 1462
query-registry-key-info 406, 1715
query-registry-value 406, 1716
quit 4, 401, 1023
read-dhparams 352, 526
read-foreign-modules 941
read-java-field 1156
read-serial-port-char 1355
read-serial-port-string 1355
reconnect 292, 1464
record-definition 696
record-java-class-lisp-symbol 1158
record-message-in-windows-event-log 1690
reduce-memory 133, 856
references-who 858
regexp-find-symbols 1026
registry-key-exists-p 406, 1717
registry-value 406, 1717
remove-advice 52, 58, 1027
remove-duplicates 98
remove-from-process-private-property 250, 1320
remove-from-process-property 250, 1320
remove-process-private-property 250, 1321
remove-process-property 250, 1322
remove-special-free-action 134, 858
remove-symbol-profile 139, 859
replace-socket-stream-socket 527
replace-standard-object 437
report-error-to-java-host 1159
reset-code-coverage 747
reset-code-coverage-snapshot 814
reset-java-interface-for-new-jvm 1160
reset-profile 139, 860
reset-ring 861
restore-code-coverage-data 747
restore-sql-reader-syntax-state 317, 1465
results for traced 38
Index

reverse-subtract-code-coverage-data 719
ring-length 861
ring-name 862
ringp 861
ring-pop 862
ring-push 863
ring-ref 864
rollback 300, 302, 329, 1465
room 127, 128, 131, 396, 614
room-values 396, 1637
rotate-ring 865
round-to-single-precision 1030
run-shell-command 1638
safe-locale-file-encoding 1643
save-argument-real-p 866
save-code-coverage-data 767
save-current-code-coverage 767
save-current-session 867
save-image 147, 159, 392, 397, 401, 868
save-image-with-bundle 877
save-tags-database 700
save-universal-from-script 878
sbchar 1031
schedule-timer 248, 1323
schedule-timer-milliseconds 1325
schedule-timer-relative 1327
schedule-timer-relative-milliseconds 1328
security-description-string-for-open-named-pipe 1692
select 298, 309, 1466
semaphore-acquire 243, 1330
semaphore-count 243, 1331
semaphore-name 243, 1332
semaphore-release 243, 1332
semaphore-wait-count 243, 1333
send-message-to-java-host 1161
sequencep 1031
serial-port 1354
serial-port-input-available-p 1356
server-terminate 528
set-application-themed 1696
set-approaching-memory-limit-callback 1644
set-array-single-thread-p 879
set-array-weak 135, 880
set-automatic-gc-callback 132, 1645
set-blocking-gen-num 132, 1647
set-clos-initarg-checking 439
set-code-coverage-snapshot 814
set-debugger-options 663
set-default-character-element-type 377, 380, 381, 1032
set-default-generation 119, 128, 881
set-default-segment-size 133, 1649
set-delay-promotion 133, 1651
setf cdr-assoc 1543
setf timer-name 1335
set-file-dates 1652
set-funcall-async-limit 1334
set-gc-parameters 121, 128, 882
set-gen-num-gc-threshold 132, 1653
set-hash-table-weak 135, 884
set-java-field 1156
set-make-instance-argument-checking 440
set-maximum-memory 128, 1654
set-maximum-segment-size 130, 132, 1655
set-memory-check 1657
set-memory-exhausted-callback 1658
set-minimum-free-space 121, 128, 886
set-prepared-statement-variables 1470
set-primitive-array-region 1103
set-process-profiling 139, 140, 887
set-profiler-threshold 139, 889
set-promotion-count 890
set-quit-when-no-windows 1033
set-registry-value 406, 1719
set-serial-port-state 1357
set-signal-handler 1660
set-spare-keeping-policy 133, 1661
set-ssl-ctx-dh 352, 531
set-ssl-ctx-options 352, 532
set-ssl-ctx-password-callback 352, 533
set-ssl-library-path 361, 534
sets-who 896
set-system-message-log 891
set-temp-directory 406, 1662
setup-atomic-funcall 1663
setup-field-accessor 1161
setup-java-caller 1163
setup-java-constructor 1163
setup-lisp-proxy 1164
set-up-profiler 138, 892
set-verification-mode 529
short-namestring 392
short-site-name 391, 620
sid-string-to-user-name 1698
simple-augmented-string-p 1535
simple-base-string-p 948
simple-bmp-string-p 951
simple-char-p 1034
simple-lock-and-condition-variable-wait 1334
simple-text-string-p 1043
single-form-form-parser 701
single-form-with-options-form-parser 702
socket-stream-address 541
socket-stream-ctx 351, 541
socket-stream-peer-address 542
socket-stream-shutdown 543
socket-stream-ssl 351, 544
software-type 391, 621
software-version 391, 622
source-debugging-on-p 896
split-sequence 1035
split-sequence-if 1036
split-sequence-if-not 1037
sql 316, 1473
sql-expression 316, 1476
sql-operation 315, 1478
sql-operator 316, 1480
sql-recording-p 319, 1481
sql-stream 319, 1482
ssl-add-client-ca 353
ssl-cipher-get-bits 353
ssl-cipher-get-name 353
ssl-cipher-get-version 353
ssl-clear-num-renegotiations 353
ssl-ctrl 353
ssl-ctx-add-client-ca 353
ssl-ctx-add-extra-chain-cert 353
ssl-ctx-ctrl 353
ssl-ctx-get-max-cert-list 353
ssl-ctx-get-mode 353
ssl-ctx-get-options 354
ssl-ctx-get-read-ahead 354
ssl-ctx-get-verify-mode 354
ssl-ctx-load-verify-locations 354
ssl-ctx-need-tmp-rsa 354
ssl-ctx-ssl-get-cache-mode 354
ssl-ctx-ssl-get-cache-size 354
ssl-ctx-ssl-get-cache-mode 354
ssl-ctx-ssl-get-cache-size 354
ssl-ctx-ssl-get-cache-mode 354
ssl-ctx-ssl-get-cache-size 354
ssl-ctx-ssl-set-cache-mode 354
ssl-ctx-ssl-set-cache-size 354
ssl-ctx-ssl-set-cache-mode 354
ssl-ctx-ssl-set-cache-size 354
ssl-ctx-ssl-set-cache-mode 354
ssl-ctx-ssl-set-cache-size 354
ssl-ctx-use-certificate-file 354
ssl-ctx-use-ssl-certicate-file 354
ssl-ctx-use-ssl-certicate-file 354
ssl-ctx-use-ssl-certicate-file 354
ssl-ctx-use-ssl-certicate-file 354
ssl-get-current-cipher 354
ssl-get-max-cert-list 354
ssl-get-mode 355
ssl-get-verify-mode 355
ssl-get-version 355
ssl-load-client-ca-file 355
ssl-need-tmp-rsa 355
ssl-new 359, 547
ssl-num-renegotiations 355
ssl-session-reused 355
ssl-set-accept-state 355, 358
ssl-set-client-ca-list 355
ssl-set-connect-state 355, 358
ssl-set-max-cert-list 355
ssl-set-mode 355
ssl-set-options 355
ssl-set-ssl-dh 355
ssl-set-ssl-rsa 355
ssl-total-renegotiations 355
ssl-use-certificate-file 355
ssl-use-privatekey-file 355
ssl-use-rsaprivatekey-file 355
standard-instance-access 214
start-dde-server 1765
start-gc-timing 897
start-profiling 139, 140, 898
start-sql-recording 319, 1484
start-tty-listener 1038
start-up-server 548
start-up-server-and-mp 554
staticp 1667
status 292, 1485
stchar 1039
stop-gc-timing 897
stop-profile 139, 140, 900
stop-sql-recording 319, 1486
string-append 1039
string-append* 1040
string-ip-address 554
structurep 1041
subtract-code-coverage-data 719
sweep-all-objects 134, 901
sweep-gen-num-objects 1668
switch-open-tcp-stream-with-ssl-to-java 555
switch-static-allocation 118, 119, 902
symbol-dynamically-bound-p 903
symeval-in-process 251, 1335
table-exists-p 1487
text-string-p 1043
throw-an-exception 1167
timer-expired-p 1336
timer-name 1337
toggle-source-debugging 99, 905
total-allocation 127, 906
traceable-dspec-p 703
trace-new-instances-on-access 443
trace-on-access 444
tracing-enabled-p 703
tracing-state 704
ture 1043
true 638
try-compact-in-generation 126, 128, 915
try-move-in-generation 126, 128, 917
typed-aref 1669
unbreak-new-instances-on-access 447
unbreak-on-access 447
unicode-alpha-char-p 389, 1045
unicode-alphanumericp 389, 1046
unicode-both-case-p 389, 1047
unicode-char-equal 389, 1048
unicode-char-greaterp 389, 1049
unicode-char-lessp 388, 1049
unicode-char-not-equal 389, 1048
unicode-char-not-greaterp 388, 1050
unicode-char-not-lessp 389, 1050
unicode-lower-case-p 389, 1051
unicode-string-equal 389, 1051
unicode-string-greaterp 389, 1053
unicode-string-lessp 389, 1053
unicode-string-not-equal 389, 1051
unicode-string-not-greaterp 390, 1054
unicode-string-not-lessp 390, 1054
unicode-upper-case-p 389, 1055
unlocked-queue-ready 836
unlocked-queue-send 836
unnotice-fd 1339
unschedule-timer 1339
untrace-new-instances-on-access 448
untrace-on-access 448
update-instance-for-different-class 640
Index

update-instance-for-redefined-class 641
update-objects-joins 1488
update-records 300, 301, 328, 1490
user-homedir-pathname 193, 404, 1577
user-name-to-sid-string 1701
user-preference 402, 1056
valid-external-format-p 718
variable-information 921
vector-pop 232
vector-push 232
vector-push-extend 232
verify-java-caller 1168
verify-java-callers 1169
verify-lisp-proxies 1172
verify-lisp-proxy 1172
wait-for-input-streams 1671
wait-for-input-streams-returning-first 1672
wait-for-wait-state-collection 556
wait-processing-events 1340
wait-serial-port-state 1357
wait-state-collection-stop-loop 557
whitespace-char-p 1060
who-binds 923
who-calls 924
who-references 924
who-sets 925
write-java-class-definitions-to-file 1174
write-java-class-definitions-to-stream 1174
write-serial-port-char 1358
write-serial-port-string 1359
yield 1347
fundamental-binary-input-stream class 1497
fundamental-binary-output-stream class 1498
fundamental-binary-stream class 1498
fundamental-character-input-stream class 340, 1499
fundamental-character-output-stream class 340, 1500
fundamental-input-stream class 1501
fundamental-output-stream class 1502

fundamental-stream class 1502

G

garbage collection, see also storage management
main chapter 111
GBK 383
:gbk external format 383
gc-generation function 118, 120, 128, 132, 815
gc-if-needed function 128, 818
general-handle-event generic function 1237
generate-code-coverage function 819
generate-java-class-definitions function 1095
generation
definition 112
generation 2 126
generation-number function 126, 1568
generic functions
accessor-method-slot-definition 214
add-method 215
class-extra-initargs 425
close 563
compute-applicable-methods-using-classes 215
compute-class-potential-initargs 426
compute-discriminating-function 215, 428
compute-effective-method-function-from-classes 428
dde-client-advise-data 1730
dde-client-advise-data 1730
dde-client-request 1753
dde-server-poke 1753
dde-server-request 1754
dde-server-topic 1755
dde-server-topics 1756
dde-topic-items 1758
default-constructor-arguments 1073
delete-instance-records 309, 1390, 1404
describe-object 26
documentation 589
general-handle-event 1237
get-inspector-values 999
input-stream-p 596
Index

instance-refreshed 309
make-instance 605
make-method-lambda 214
open-stream-p 611
output-stream-p 611
print-object 157
process-a-class-option 433
process-a-slot-option 435
slot-boundp-using-class 213, 440
slot-makunbound-using-class 213, 441
slot-value-using-class 213, 442
sort-inspector-p 999
stream-advance-to-column 1503
stream-check-eof-no-hang 1504
stream-clear-input 342, 1504
stream-clear-output 343, 1505
stream-element-type 340, 627
stream-file-position 1506
stream-fill-buffer 1506
stream-finish-output 343, 1507
stream-flush-buffer 1508
stream-force-output 343, 1509
stream-fresh-line 1509
stream-line-column 343, 1510
stream-listen 342, 1511
stream-output-width 1512
stream-peek-char 1512
stream-read-buffer 1513
stream-read-byte 1514
stream-read-char 341, 1515
stream-read-char-no-hang 1515
stream-read-line 1516
stream-read-sequence 1517
stream-read-timeout 1518
stream-start-line-p 343, 1518
stream-terpri 1519
stream-unread-char 341, 1520
stream-write-buffer 1520
stream-write-byte 1521
stream-write-char 342, 1522
stream-write-sequence 1522
stream-write-string 1523
update-instance-from-records 309, 1487
update-record-from-instance 1491
update-record-from-slot 309, 1491
update-records-from-instance 309
update-slot-from-record 309, 1492

wait-for-connection 1701

:gen-num initarg 1668
gen-num-segments-fragmentation-state function 132, 1566
gensym function 134
gesture-spec-accelerator-bit variable 1568
gesture-spec-caps-lock-bit variable 1568
gesture-spec-control-bit variable 1568
gesture-spec-data function 1570
gesture-spec-hyper-bit variable 1568
gesture-spec-meta-bit variable 1568
gesture-spec-modifiers function 1571
gesture-spec-p function 1571
gesture-spec-shift-bit variable 1568
gesture-spec-super-bit variable 1568
gesture-spec-to-character function 1572
get 729
:get listener command 6
getApplicationContext Java method 1201
getClassLoader Java method 1201
get-code-coverage-delta function 814
get-current-process function 226, 227, 1238
get-default-generation function 128, 822
get-default-local-ipv6-address function 501
get-file-stat function 1573
get-folder-path function 405, 1575
get-foreign-symbol function 939
get-form-parser function 691
get-gc-parameters function 128, 822
get-gc-timing function 897
gethash-ensuring function 825
get-host-entry function 502
get-inspector-values generic function 999
get-ip-default-zone-id function 505
get-java-virtual-machine function 1101
get-jobobject function 1102
get-primitive-array-region function 1103
GetProcAddress C function 159
get-process function 227, 1239
get-process-private-property function 1240
get-socket-address function 507
get-socket-peer-address function 508
get-superclass-and-instances-tree function 1105
get-temp-directory function 824
get-verification-mode function 824
get-working-directory function 824

grammar
  non-terminal 268
  resolving ambiguities 268

graphics ports xlix
  > SQL operator 312
*grep-command* variable 1002
*grep-command-format* variable 1003
*grep-fixed-args* variable 1004

guess-external-format function 386, 1579

GUI application 869

H
  :h inspector command 27
*handle-existing-action-in-action-list* variable 1004, 78
*handle-existing-action-list* variable 1005, 78
*handle-existing-defpackage* variable 827

*handle-missing-action-in-action-list* variable 1006, 79
*handle-missing-action-list* variable 1005, 79
*handle-old-in-package* variable 828
*handle-old-in-package-used-as-make-package* variable 829

handler frame, examining 12
:handler keyword 659
*handle-warn-on-redefinition* variable 155, 1006

hardcopy-system function 1007

hash tables
  weak 602
  heap size
    in 32-bit LispWorks 398, 416
    in 64-bit LispWorks 398, 416

:help listener command 7

HFS+ filesystem 407
*hiden* keyword 651
*hidden-packages* variable 24, 650
:his listener command 7

hook functions 77

host 502
host name 502
hostname 502

I
  :i inspector command 27

i18n 373
image
  saving 147
  image size 395
immediatep function 1580
impersonating-named-pipe-client macro 1681

impersonating-user macro 1682
import-java-class-definations macro 1106

incf macro 231

init file 1008
init Java method 1187
init_result_code Java method 1190
*init-file-name* variable 1008

initialization
  of Common SQL 288
  initialization file 1008

initialize-database-type function 1401
Index

*initialized-database-types* variable 1402
initialize-multiprocessing function 227, 1240
*initial-processes* variable 163, 227, 1242
init-java-interface function 1108
InitLispWorks C function 162, 397, 1767
input-stream-p generic function 596
insert-records function 300, 328, 1402
:inside keyword 43
inspect function 26 inspector
main chapter 25 REPL 25 teletype 25
inspector commands
:cv 27
:d 27
:dm 27
:dr 27
:fh 27
:i 27
:m 27, 28
:q 27
:s 27
:sh 27
:u 27
:ud 27, 28
*inspect-print-length* variable 27
*inspect-print-level* variable 27
*inspect-through-gui* variable 1009
instance-refreshed generic function 309
in-static-area macro 119, 1580
int32* function 1582
int32+ function 411, 1582
int32- function 411, 1582
int32 type 410, 1581
int32/ function 1582
int32/= function 1585
int32<- function 1585
int32<< function 1586
int32<= function 1585
int32=> function 1585
int32>= function 1585
int32>> function 1586
+int32-0+ symbol macro 1583
int32-1+ function 1584
int32-1- function 1584
+int32-1+ symbol macro 1583
int32-aref function 1587
int32-logand function 1588
int32-logandcl function 1588
int32-logandc2 function 1588
int32-logbitp function 1588
int32-logeqv function 1588
int32-logior function 1588
int32-lognand function 1588
int32-lognor function 1588
int32-lognot function 1588
int32-logorc1 function 1588
int32-logorc2 function 1588
int32-logtest function 1588
int32-logxor function 1588
int32-minusp function 1590
int32-plusp function 1590
int32-to-int64 function 1591
int32-to-integer function 1592
int32-zerop function 1590
int64* function 1593
int64+ function 412, 1593
int64- function 412, 1593
int64 type 412, 1592
int64/= function 1593
int64//= function 1596
int64< function 1596
int64<< function 1597
int64<= function 1596
int64>= function 1596
int64>> function 1597
+int64-0+ symbol macro 1594
int64-1+ function 1595
int64-1- function 1595
+int64-1+ symbol macro 1594
int64-aref function 1598
int64-logand function 1599
int64-logandcl function 1599
int64-logandc2 function 1599
int64-logbitp function 1599
int64-logeqv function 1599
int64-logior function 1599
int64-lognand function 1599
int64-lognor function 1599
int64-lognot function 1599
Index

java-array-length function 1118
java-array-simple-error
condition 1118
java-bad-jobject condition 1119
java-bad-jobject-caller accessor 1119
java-bad-jobject-object accessor 1119
java-class-error condition 1120
java-definition-error
condition 1120
java-definition-error-class-name accessor 1120
java-definition-error-name accessor 1120
java-exception condition 1121
java-exception-exception-name accessor 1121
java-exception-java-backtrace accessor 1121
java-exception-string accessor 1121
java-field-class-name-for-setting function 1156
java-field-error condition 1120
java-field-error-field-name accessor 1120
java-field-error-static-p accessor 1120
java-field-setting-error condition 1122
java-field-setting-error-class-name accessor 1122
java-field-setting-error-class-name-for-setting accessor 1122
java-field-setting-error-field-name accessor 1122
java-field-setting-error-new-value accessor 1122
java-id-exception condition 1123
java-instance-jobect accessor 185, 1165
java-instance-without-jobect-error condition 1123
java-interface-error condition 1124
java-low-level-exception condition 1125
java-method-error condition 1120
java-method-error-args-num accessor 1120
java-method-error-method-name accessor 1120
java-method-exception
condition 1125
java-method-exception-args accessor 1125
java-method-exception-class-name accessor 1125
java-method-exception-method-name accessor 1125
java-method-exception-name accessor 1125
java-normal-exception condition 1126
java-not-a-java-object-error condition 1127
java-not-an-array-error condition 1127
java-object-array-element-type function 1128
java-objects-eq function 1129
java-out-of-bounds-error condition 1130
java-primitive-array-element-type function 1130
java-serious-exception condition 1131
java-storing-wrong-type-error condition 1130
java-type-to-lisp-array-type function 1132
java-vm-poi type 1133
jboolean type 1134
jbyte type 1134
jchar type 1134
jdouble type 1134
jfloat type 1134
jint type 1134
JIS 382
:jis external format 382
jlong type 1134
JNI_CreateJavaVM 1109, 1111
JNI_GetCreatedJavaVMs 1101, 1111
JNI_OnLoad 184, 1111
jni-env-poi type 1134
:object initarg 1165
jobject type 1135
jobject-class-name function 1135
jobject-ensure-global function 1136
jobject-of-class-p function 1137
jobject-p function 1138
Index

jobject-pretty-class-name function 1139
jobject-string function 1139
jobject-to-lisp function 1140
join slot 306
jshort type 1134
jvref function 1141

K
keywords
:after 38
:all 263
:allocation 43
:backtrace 40
:before 37
:bindings 655
:break 39
:break-on-exit 39
:catchers 658
:caused-by 263
:default-pathname 261
:dont-know 17
:entrycond 40
:eval-after 39
:eval-before 38
:exitcond 40
:handler 659
:hidden 651
:inside 43
:invisible 660
:maxium-overflow 125
:members 261
:minimum-for-sweep 121, 125
:minimum-overflow 125
:new-generation-size 126
:package 261
:previous 263
:process 42
:requires 263, 264
:restarts 662
:rules 262
:source-only 262
:step 40
:trace-output 42
:when 43
KnowledgeWorks rules
compiling dynamically 85
known-sid-integer-to-sid-string function 1683
KO18 383
:koi-8 external format 383
KO18-R 383

L
:l debugger command 17
:lambda debugger command 19
last-callback-on-thread function 1242
Latin-1 381
:latin-1 external format 381
*latin-1-code-pages* variable 1684
:latin-1-safe external format 381
:latin-1-terminal external format 381
levels of safety, see compiler
:if debugger command 19
library formats 941
*line-arguments-list* variable 392, 1604
lisp image
filename 392, 1009
pathname 392, 1009
lisp-array-to-primitive-array function 1153
lisp-array-type-to-java-type function 1132
LispCalls Java class 1179
LispErrorReporter Java interface 1193
LispGuiErrorReporter Java interface 1193
lisp-image-name function 392, 1009
lisp-java-instance-p function 1143
lisp-name-to-foreign-name function 940
lisp-to-jobbect function 1144
LispWorks
customizing 146
processes 225
quitting 4, 81
saving 2
starting 1, 81
threads in 225
LispWorks as a DLL 159
LispWorks as a dynamic library 159
LispWorks as a shared library 159
LispWorks IDE
Debugger tool 985
Help menu 213, 561
Inspector tool 1009
Notifier window 985

1807
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>lispworks-directory</em> variable 1010</td>
</tr>
<tr>
<td>LispWorksDlsym C function 1770</td>
</tr>
<tr>
<td>LispWorksState C function 162, 1771</td>
</tr>
<tr>
<td>list-all-processes function 227, 1243</td>
</tr>
<tr>
<td>list-attributes function 305, 1406</td>
</tr>
<tr>
<td>list-attribute-types function 305, 1405</td>
</tr>
<tr>
<td>list-classes function 309, 1407</td>
</tr>
<tr>
<td>listener 1038</td>
</tr>
<tr>
<td>top level commands 1552</td>
</tr>
<tr>
<td>listener commands</td>
</tr>
<tr>
<td>:? 7</td>
</tr>
<tr>
<td>:bug-form 7</td>
</tr>
<tr>
<td>:get 6</td>
</tr>
<tr>
<td>:help 7</td>
</tr>
<tr>
<td>:his 7</td>
</tr>
<tr>
<td>:redo 6</td>
</tr>
<tr>
<td>:use 7</td>
</tr>
<tr>
<td>listener process 1242</td>
</tr>
<tr>
<td>listener prompt 1022</td>
</tr>
<tr>
<td>list-sql-streams function 319, 1408</td>
</tr>
<tr>
<td>list-tables function 305, 1408</td>
</tr>
<tr>
<td>load-all-patches function 1012</td>
</tr>
<tr>
<td>load-code-coverage-data function 767</td>
</tr>
<tr>
<td>load-data-file function 830</td>
</tr>
<tr>
<td><em>load-fasl-or-lisp-file</em> variable 833</td>
</tr>
<tr>
<td>LoadLibrary C function 159</td>
</tr>
<tr>
<td>loadLibrary Java method 1192</td>
</tr>
<tr>
<td>load-logical-pathname-translations function 598</td>
</tr>
<tr>
<td>load-on-demand 156</td>
</tr>
<tr>
<td><em>load-source-if-newer</em> function 1014</td>
</tr>
<tr>
<td>load-system function 1013</td>
</tr>
<tr>
<td>:lob-locator initarg 1409</td>
</tr>
<tr>
<td>lob-stream class 327, 331, 336, 1409</td>
</tr>
<tr>
<td>lob-stream-lob-locator accessor 1410</td>
</tr>
<tr>
<td>lock-and-condition-variable-broadcast function 1244</td>
</tr>
<tr>
<td>lock-and-condition-variable-signal function 1245</td>
</tr>
<tr>
<td>lock-and-condition-variable-wait function 1246</td>
</tr>
<tr>
<td>lock-locked-p function 1248</td>
</tr>
<tr>
<td>lock-name function 232, 1250</td>
</tr>
<tr>
<td>lock-owned-by-current-process-p function 1248</td>
</tr>
<tr>
<td>lock-owner function 232, 1251</td>
</tr>
<tr>
<td>lock-recursively-locked-p function 1250</td>
</tr>
<tr>
<td>lock-recurse-p function 1249</td>
</tr>
<tr>
<td>locks 232</td>
</tr>
<tr>
<td>log-bug-form function 652</td>
</tr>
<tr>
<td>logs-directory function 653</td>
</tr>
<tr>
<td>long-float type 598</td>
</tr>
<tr>
<td>long-namestring function 392, 1685, 1697</td>
</tr>
<tr>
<td>long-site-name function 391, 599</td>
</tr>
<tr>
<td>loop macro 297, 302, 599, 1412</td>
</tr>
<tr>
<td>extensions in Common SQL 310</td>
</tr>
<tr>
<td>loop, extensions in Common SQL 303</td>
</tr>
<tr>
<td>loop-processing-wait-state-collection function 513</td>
</tr>
<tr>
<td>Low level atomic operations 253</td>
</tr>
<tr>
<td>low-level-atomic-place-p function 1606</td>
</tr>
<tr>
<td>lpcstr FLI type 1698</td>
</tr>
<tr>
<td>lpcstr FLI type 1699</td>
</tr>
<tr>
<td>lpcwstr FLI type 1700</td>
</tr>
<tr>
<td>lpstr FLI type 1698</td>
</tr>
<tr>
<td>lptstr FLI type 1699</td>
</tr>
<tr>
<td>lpwstr FLI type 1700</td>
</tr>
<tr>
<td>M</td>
</tr>
<tr>
<td>:m: inspector command 27, 28</td>
</tr>
<tr>
<td>Mach-O bundle 873</td>
</tr>
<tr>
<td>Mach-O dynamically linked shared library 871</td>
</tr>
<tr>
<td>:macos-roman external format 382</td>
</tr>
<tr>
<td>macros</td>
</tr>
<tr>
<td>advice 53</td>
</tr>
<tr>
<td>allocation-in-gen-num 119, 128, 723</td>
</tr>
<tr>
<td>allowing-block-interrupts 245, 1205</td>
</tr>
<tr>
<td>analysing-special-variables-usage 724</td>
</tr>
<tr>
<td>appendf 945</td>
</tr>
<tr>
<td>at-location 672</td>
</tr>
</tbody>
</table>
Index

without-code-coverage 935
without-interrupts 245, 1345
without-preemption 245, 1346
with-output-to-fasl-file 930
with-output-to-string 641
with-registry-key 406, 1720
with-ring-locked 934
with-sharing-lock 233, 1344
with-stream-input-buffer 1524
with-stream-output-buffer 1526
with-transaction 300, 301, 1493
with-unique-names 1063
with-windows-event-log-event-source 1702
mailbox-count function 1252
mailbox-empty-p function 1253
mailbox-not-empty-p function 1253
mailbox-peek function 1254
mailbox-read function 1255
mailbox-reader-process function 1256
mailbox-send function 1257
mailbox-wait function 1257
mailbox-wait-for-event function 1259
*main-process* variable 1261
make-array function 135, 232, 601
make-barrier function 242, 1261
make-condition-variable function 1262
make-gesture-spec function 1607
make-hash-table function 135, 232, 602
make-instance generic function 605
make-java-array function 1144
make-java-instance function 1145
make-lisp-proxy function 1146
make-lisp-proxy-with-overrides function 1146
make-lock function 1263
make-mailbox function 1265
make-method-lambda generic function 214
make-mt-random-state function 1016
make-named-timer function 1266
make-ring function 834
make-semaphore function 243, 1267
make-sequence function 606
make-simple-int32-vector function 1612
make-simple-int64-vector function 1613
make-ssl-ctx function 356, 357, 512
make-stderr-stream function 1614
make-string-output-stream function 606
make-symbol function 134
make-timer function 248, 1268
make-typed-aref-vector function 1615
make-unlocked-queue function 836
make-unregistered-action-list function 1014
make-wait-state-collection function 514
Manager Java class 1185
makeجمال function 607
map-all-processes function 1269
map-all-processes-backtrace function 1270
map-environment function 1616
map-java-object-array function 1149
map-process-backtrace function 1270
map-processes function 1271
map-query function 302, 328, 1414
map-ring function 837
mark and sweep 120
mark-and-sweep function 838
marking-gc function 132, 1617
*maximum-ordinary-windows* variable 154
:maximum-overflow keyword 125
*max-trace-indent* variable 46, 840
:members keyword 261
memory allocation during tracing 43
memory clashes 162
avoiding 396
memory management common features 133
garbage collection strategy in 32-bit LispWorks 121
image reduction 133
image relocation 395
in 32-bit LispWorks 117
in 64-bit LispWorks 128
mark and sweep 120
overflow 125
relocating the image 395
timing in 115, 131, 133
memory-growth-margin
  function 128, 1619
merge function 608
merge-code-coverage-data
  function 841
merge-ef-specs function 1620
Mersenne Twister 1017
MessageHandler Java interface 1200
Metaobject Protocol 213
metaobject protocol
  class options 433
  slot options 436
method
  advice 53
method-combination class 215
methods
  tracing 45
:minimum-for-sweep keyword 121, 125
:minimum-overflow keyword 125
mInitErrorString Java method 1192
minus SQL operator 312
mMaxErrorLogsNumber Java
  field 1196
mMessagesMaxLength Java field 1198
mod 2^32 arithmetic 410
mod 2^64 arithmetic 412
modify-hash function 230, 231, 842
modifying a database 300
MOP
  AMOP compatibility 213
  class options 433
  slot options 436
most-positive-fixnum
  constant 417
mt-random function 1017
mt-random-state type 1018
*mt-random-state* variable 1017
mt-random-state-p function 1018
*multibyte-code-page-ef*
  variable 337, 1686
multi-processing
  locks 232
multiprocessing
  locks 232
:multiprocessing delivery
  keyword 228
MySQL
  connecting 293
MySQL client library 295
Mac OS X 295
*mysql-library-directories*
  variable 295, 296, 1415
*mysql-library-path*
  variable 295, 296, 1416
*mysql-library-sub-directories*
  variable 295, 1417
N
:n debugger command 16
name-defined-dspecs function 693
name-definition-locations
  function 694
named-pipe-stream-name
  function 1686
name-only-form-parser
  function 695
New in LispWorks 7.0
accepting-handle type 454
accepting-handle-collection
  function 455
accepting-handle-local-port
  function 456
accepting-handle-name
  function 456
accepting-handle-socket
  function 457
accepting-handle-user-info
  function 458
accept-tcp-connections-creating-async-io-states
  function 451
add-code-coverage-data
  function 719
addMessage Java method 1198
AIX PowerPC/32-bit LispWorks 571
AIX PowerPC/64-bit LispWorks 571
:android-delivery feature 592
android-funcall-in-main-thread
  function 728
android-funcall-in-main-thread-list
  function 728
android-get-current-activity
  function 729
*android-main-process-for-testing*
  variable 730
android-main-thread-p
  function 731
apply-in-wait-state-collection-process
  function 458
approaching-memory-limit

1811
Index

condition 1530
:arm feature 592
ARM Linux/32-bit LispWorks 571
Asynchronous I/O 361
async-io-state structure type 459
async-io-state-abort function 460
async-io-state-abort-and-close function 462
async-io-state-address function 463
async-io-state-buffered-data-length function 464
async-io-state-discard function 464
async-io-state-finish function 465
async-io-state-get-buffered-data function 466
async-io-state-peer-address function 467
async-io-state-read-buffer function 468
async-io-state-receive-message function 473
async-io-state-send-message function 475
async-io-state-send-message-to-address function 477
async-io-state-write-buffer function 479
*background-input* variable 736
*background-output* variable 736
*background-query-io* variable 736
backlog argument to accept-tcp-connections-creating-asyn-co- states 452
backlog argument to start-up-server 548
base-char-ref function 1623
:bmp external format 707
bmp-char type 948
bmp-char-p function 949
:bmp-native external format 707
:bmp-reversed external format 707
bmp-string type 950
bmp-string-p function 951
brackets-limits argument to find-regexp-in-string 996
BugFormLogsList Java class 1202
BugFormViewer Java class 1203
callDoubleA Java method 1180
callDoubleV Java method 1180
callIntA Java method 1180
callIntV Java method 1180
call-java-method function 1065
call-java-method-error condition 1066
callObjectA Java method 1180
callObjectV Java method 1180
callVoidA Java method 1180
callVoidV Java method 1180
call-wait-state-collection function 482
catching-exceptions-bind macro 1067
catching-java-exceptions macro 1067
checked-read-java-field function 1156
check-java-field function 1156
check-lisp-calls-initialized function 1069
checkLispSymbol Java method 1183
clearBugFormLogs Java method 1196
clear-code-coverage function 747
close-accepting-handle function 482
close-async-io-state function 483
close-wait-state-collection function 484
code-coverage-data type 748
code-coverage-data-create-time accessor 749
code-coverage-data-coverage-html function 750
code-coverage-data-coverage-statistics function 756
code-coverage-data-name accessor 749
code-coverage-file-stats type 757
code-coverage-file-stats-called accessor 757
code-coverage-file-stats-counters-count accessor 757
code-coverage-file-stats-
Index

counters-executed accessor 757
code-coverage-file-stats-counters-hidden accessor 757
code-coverage-file-stats-fully-covered accessor 757
code-coverage-file-stats-hidden-covered accessor 757
code-coverage-file-stats-lambdas-count accessor 757
code-coverage-file-stats-not-called accessor 757
code-coverage-file-stats-partially-covered accessor 757
code-coverage-file-stats-source-file accessor 757
code-coverage-set-editor-default-data function 761
code-coverage-set-html-background-colors function 762
compute-effective-method-function-from-classes generic function 428
Conditional throw and checking for catch in the dynamic environment 414

copy-code-coverage-data function 767
copy-current-code-coverage function 767
copy-standard-object function 431
create-and-run-wait-state-collection function 484
create-async-io-state function 487
create-async-io-state-and-connected-tcp-socket function 493
create-async-io-state-and-connected-udp-socket function 491
create-async-io-state-and-udp-socket function 493
create-instance-from-job function 1069
create-instance-object function 1070
create-instance-object-list function 1070
create-java-object function 1072
create-java-object-error condition 1073
createLispProxy Java method 1182
current-process-kill function 1226
current-process-send function 1229
current-process-set-terminate-method function 1230
decode-to-db-standard-date function 1380
decode-to-db-standard-timestamp function 1380
default-constructor-arguments generic function 1073
default-name-constructor function 1074
define-field-accessor macro 1075
define-java-caller macro 1076
define-java-callers macro 1079
define-java-constructor macro 1076
define-lisp-proxy macro 1081
delivered-image-p function 782
deliver-to-android-project function 783
destructive-add-code-coverage-data function 719
destructive-merge-code-coverage-data function 841
destructive-reverse-subtract-code-coverage-data function 719
destructive-subtract-code-coverage-data function 719
detect-utf32-bom function 1556
detect-utf8-bom function 1556
dump-forms-to-file additional arguments and improved documentation 930
editor-color-code-coverage function 791
:embedded-module member option for defsystem :type :c-file 972
encode-db-standard-date function 1381
encode-db-standard-timestamp function 1381
ensure-hash-entry function 796
ensure-is-jobject function 1102
ensure-lisp-classes-from-tree function 1089
ensure-supers-contain-java.lang.object function 1091
:error argument in gesture-spec-to-character 1572
errorp argument to gesture-spec-to-character 1572
error-situation-forms macro 797
example-edit-file function 989
fast-error condition 802
fast-directory-files function 803
fdf-handle-directory-p accessor 803
fdf-handle-directory-string accessor 803
fdf-handle-last-access accessor 803
fdf-handle-last-modify accessor 803
fdf-handle-link-p accessor 803
fdf-handle-size accessor 803
fdf-handle-writable-p accessor 803
field-access-exception condition 1092
field-access-exception-set-p accessor 1092
field-exception condition 1093
field-exception-class-name accessor 1093
field-exception-field-name accessor 1093
filter-code-coverage-data function 807
find-coding-option supports GNU Emacs coding option 1565
find-java-class function 1094
find-throw-tag function 809
format-to-java-host function 1095
funcall-async function 1236
funcall-async-list function 1236
generate-code-coverage function 819
generate-java-class-definitions function 1095
gesture-spec-caps-lock-bit constant 1568
getApplicationContext Java method 1201
classloader function 1201
get-code-coverage-delta function 814
get-default-local-ipv6-address function 501
get-folder-path is now available on all platforms 1575
get-gc-timing function 897
gethash-ensuring function 825
get-ip-default-zone-id function 505
get-java-virtual-machine function 1101
get-jobject function 1102
get-primitive-array-region function 1103
get-service-entry function 505
get-superclass-and-interfaces-tree function 1103
immediatep function 1580
import-java-class-definitions macro 1106
init Java method 1187
init_result_code Java method 1190
init-java-interface function 1198
int32-to-int64 function 1591
int64* function 1593
int64+ function 1593
int64- function 1593
int64 type 1592
int64/ function 1593
int64/= function 1596
int64<= function 1596
int64<= function 1597
int64<= function 1596
int64> function 1596
int64> function 1596
int64> function 1597
+int64-0+ symbol macro 1594
int64-1+ function 1595
Index

int64-1-function 1595
+int64-1+ symbol macro 1594
int64-aref function 1598
int64-logand function 1599
int64-logandc1 function 1599
int64-logandc2 function 1599
int64-logbtp function 1599
int64-logeqv function 1599
int64-logior function 1599
int64-lognand function 1599
int64-lognot function 1599
int64-logorc1 function 1599
int64-logorc2 function 1599
int64-logtest function 1599
int64-logxor function 1599
int64-plusp function 1601
int64-to-int32 function 1602
int64-to-integer function 1603
int64-zerop function 1601
integer-to-int64 function 1604
intern-and-export-list
  function 1113
:ios-delivery feature 592
jaref function 1114
java-array-element-type
  function 1115
java-array-error condition 1116
java-array-error-array
  accessor 1117
java-array-error-caller
  accessor 1117
java-array-indices-error
  condition 1117
java-array-indices-error-indices
  accessor 1117
java-array-indices-error-rank
  accessor 1117
java-array-length function 1118
java-array-simple-error
  condition 1118
java-bad-jobect condition 1119
java-bad-jobect-caller
  accessor 1119
java-bad-jobect-object
  accessor 1119
java-class-error condition 1120
java-definition-error
  condition 1120
java-definition-error-class-name
  accessor 1120
java-definition-error-name
  accessor 1120
java-exception condition 1121
java-exception-exception-name
  accessor 1121
java-exception-java-backtrace
  accessor 1121
java-exception-string
  accessor 1121
java-field-class-name-for-setting
  function 1156
java-field-error condition 1120
java-field-error-field-name
  accessor 1120
java-field-error-static-p
  accessor 1120
java-field-setting-error
  condition 1122
java-field-setting-error-class-name
  accessor 1122
java-field-setting-error-class-name-for-setting
  accessor 1122
java-field-setting-error-field-name
  accessor 1122
java-id-exception
  condition 1123
java-instance-jobect
  accessor 1165
java-instance-without-job-ject-error
  condition 1123
java-interface-error
  condition 1124
java-low-level-exception
  condition 1125
java-method-error
  condition 1120
java-method-error-args-num
  accessor 1120
java-method-error-method-name
  accessor 1120
java-method-exception
  condition 1125
java-method-exception-args
  accessor 1125
java-method-exception-class-name
  accessor 1125
java-method-exception-method-name
  accessor 1125
java-method-exception-name
  accessor 1125

1815
Index

java-normal-exception condition 1126
java-not-a-java-object-error condition 1127
java-not-an-array-error condition 1127
java-object-array-element-type function 1128
java-objects-eq function 1129
java-out-of-bounds-error condition 1130
java-primitive-array-element-type function 1130
java-serious-exception condition 1131
java-storing-wrong-type-error condition 1130
java-type-to-lisp-array-type function 1132
java-vm-poi FLI type descriptor 1133
jboolean FLI type descriptor 1134
jbyte FLI type descriptor 1134
jchar FLI type descriptor 1134
jdouble FLI type descriptor 1134
jfloat FLI type descriptor 1134
jint FLI type descriptor 1134
jni-env-poi FLI type descriptor 1134
jobject FLI type descriptor 1135
jobject-class-name function 1135
jobject-ensure-global function 1136
jobject-of-class-p function 1137
jobject-p function 1138
jobject-pretty-class-name function 1139
jobject-string function 1139
jobject-to-lisp function 1140
jshort FLI type descriptor 1134
jvref function 1141
KOI8-R external format 383
lisp-array-to-primitive-array function 1153
lisp-array-type-to-java-type function 1132
LispCalls Java class 1179
LispErrorReporter Java interface 1193
LispGuiErrorReporter Java interface 1193
lisp-java-instance-p function 1143
lisp-to-object function 1144
load-code-coverage-data function 767
loadLibrary Java method 1192
load-processing-wait-state-collection function 513
mailbox-wait function 1257
make-java-array function 1144
make-java-instance function 1145
make-lisp-proxy function 1146
make-lisp-proxy-with-overrides function 1146
make-ring function 834
make-simple-int64-vector function 1613
make-unlocked-queue function 836
make-wait-state-collection function 514
Manager Java class 1185
map-java-object-array function 1149
map-ring function 837
merge-code-coverage-data function 841
MessageHandler Java interface 1200
mInitErrorString Java method 1192
mMaxErrorLogsNumber Java field 1196
mMessagesMaxLength Java field 1198
object-pointer function 1622
coctet-ref function 1623
open-tcp-stream-using-java function 518
Package change and additional functionality for load-data-file 830
package-flagged-p function 1629
pipe-exit-status function 1630
position-in-ring function 846
position-in-ring-forward function 846
primitive-array-to-lisp-array function 1153
Index

process-interrupt-list  function 1281
process-terminate  function 1302
read-java-field  function 1156
record-java-class-lisp-symbol  function 1158
record-message-in-windows-event-log  function 1690
replacement-source-form  macro 699
replace-socket-stream-socket  function 527
replace-standard-object  function 437
report-error-to-java-host  function 1159
reset-code-coverage  function 747
reset-code-coverage-snapshot  function 814
reset-java-interface-for-new-jvm  function 1160
reset-ring  function 861
restore-code-coverage-data  function 747
reverse-subtract-code-coverage-data  function 719
ring-length  function 861
ring-name  function 862
ringp  function 861
ring-pop  function 863
ring-push  function 863
ring-ref  function 864
rotate-ring  function 865
run-shell-command  is implemented on Windows 1638
save-code-coverage-data  function 767
save-current-code-coverage  function 767
save-exit-status  argument to open-pipe 1625
send-message-to-java-host  function 1161
sequencep  function 1031
set-approaching-memory-limit-callback  function 1644
set-code-coverage-snapshot  function 814
setCurrentActivity  Java  method 1202
setErrorReporter  Java  method 1193
set-funcall-async-limit  function 1334
setGuiErrorReporter  Java  method 1193
set-java-field  function 1156
setMessageHandler  Java  method 1200
setup-field-accessor  function 1161
setup-java-caller  function 1163
setup-java-constructor  function 1163
setup-lisp-proxy  function 1164
showBugFormLogs  Java  method 1197
simple-bmp-string  type 950
simple-bmp-string-p  function 951
simple-int64-vector  type 1665
socket-stream-shutdown  function 543
standard-java-object  class 1165
start-gc-timing  function 897
status  Java  method 1189
stop-gc-timing  function 897
string-append*  function 1040
structurep  function 1041
subtract-code-coverage-data  function 719
suspend  argument to stop-profiling 900
switch-open-tcp-stream-with-ssl-to-java  function 555
symbol-dynamically-bound-p  function 903
Test for dynamic bindings 414
The HTML version of the CLOS Metaobject protocol is available via the Help menu. 213
throw-an-exception  function 1167
throw-if-tag-found  macro 904
unlocked-queue  type 918
unlocked-queue-read  function 836
unlocked-queue-ready

1817
Index

function 836

unlocked-queue-send function 836

User Guide chapter "Code Coverage" 101

User Guide section "Code signing in saved images" 149

User Guide section "Specifying the target for connecting and binding a socket" 348

value :delete for keyword argument :load of compile-file 567

value :none for direction argument to open-pipe 1625

verify-java-caller function 1168

verify-java-callers function 1169

verify-lisp-proxies function 1172

verify-lisp-proxy function 1172

wait-for-wait-state-collection function 556

wait-state-collection class 557

wait-state-collection-stop-loop function 557

with-code-coverage-generation macro 926

with-ensuring-gethash macro 926

without-code-coverage macro 935

with-output-to-fasl-file additional arguments and improved documentation 930

with-ring-locked macro 934

with-windows-event-log-event-source macro 1702

write-java-class-definitions-to-file function 1174

write-java-class-definitions-to-stream function 1174

:new-generation-size keyword 126

Newly documented in LispWorks 7.0

sort-inspector-p generic function 999

non-terminal in grammar 268

normal-gc function 128, 843

not SQL operator 312

notice-fd function 1272

O

object

object-oriented interface in Common SQL 305

static 119

object finalization 134

Object Oriented DDL in Common SQL 306

Object Oriented DML in Common SQL 309

object-address function 1621

object-pointer function 1622

octet-ref function 1623

ODBC

connecting 293

OODDL 306

OODML 309

open function 609

opening a URL 1629

open-named-pipe-stream function 1687

open-pipe function 1624

open-registry-key function 406, 1714

open-stream-p generic function 611

open-tcp-stream function 351, 356, 514

open-tcp-stream-using-java function 518

open-temp-file function 405, 773

open-url function 1629

operating system 391

optimization

fast 32-bit arithmetic 410

fast 64-bit arithmetic 412

floating point 94

foreign slot access 98

do compiler 86

tail call 95, 143

optimization declarations 87

optimization hints 93, 573

optimize 87

optimize qualities 87–90

:optimize-slot-access class option 214, 217, 442, 579

Oracle

connecting 292

Oracle Call Interface

in Common SQL 292
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ora-lob-append</td>
<td>335, 1418</td>
</tr>
<tr>
<td>ora-lob-assign</td>
<td>334, 1419</td>
</tr>
<tr>
<td>ora-lob-char-set-form</td>
<td>333, 1419</td>
</tr>
<tr>
<td>ora-lob-close</td>
<td>1420</td>
</tr>
<tr>
<td>ora-lob-copy</td>
<td>335, 1422</td>
</tr>
<tr>
<td>ora-lob-create-empty</td>
<td>328, 334, 1423</td>
</tr>
<tr>
<td>ora-lob-create-temporary</td>
<td>336, 1424</td>
</tr>
<tr>
<td>ora-lob-disable-buffering</td>
<td>336, 1425</td>
</tr>
<tr>
<td>ora-lob-element-type</td>
<td>333, 1426</td>
</tr>
<tr>
<td>ora-lob-enable-buffering</td>
<td>336, 1427</td>
</tr>
<tr>
<td>ora-lob-env-handle</td>
<td>331, 1428</td>
</tr>
<tr>
<td>ora-lob-erase</td>
<td>335, 1428</td>
</tr>
<tr>
<td>ora-lob-file-close</td>
<td>335, 1429</td>
</tr>
<tr>
<td>ora-lob-file-close-all</td>
<td>335, 1430</td>
</tr>
<tr>
<td>ora-lob-file-exists</td>
<td>1431</td>
</tr>
<tr>
<td>ora-lob-file-get-name</td>
<td>1432</td>
</tr>
<tr>
<td>ora-lob-file-is-open</td>
<td>1432</td>
</tr>
<tr>
<td>ora-lob-file-open</td>
<td>335, 1433</td>
</tr>
<tr>
<td>ora-lob-file-set-name</td>
<td>335, 1434</td>
</tr>
<tr>
<td>ora-lob-flush-buffer</td>
<td>336, 1435</td>
</tr>
<tr>
<td>ora-lob-free</td>
<td>334, 1436</td>
</tr>
<tr>
<td>ora-lob-free-temporary</td>
<td>336, 1437</td>
</tr>
<tr>
<td>ora-lob-get-buffer</td>
<td>331, 1431</td>
</tr>
<tr>
<td>ora-lob-get-chunk-size</td>
<td>334, 1440</td>
</tr>
<tr>
<td>ora-lob-get-length</td>
<td>334, 1440</td>
</tr>
<tr>
<td>ora-lob-internal-lob-p</td>
<td>333, 1441</td>
</tr>
<tr>
<td>ora-lob-is-equal</td>
<td>334, 1442</td>
</tr>
<tr>
<td>ora-lob-is-open</td>
<td>334, 1442</td>
</tr>
<tr>
<td>ora-lob-is-temporary</td>
<td>334, 1443</td>
</tr>
<tr>
<td>ora-lob-load-from-file</td>
<td>335, 1444</td>
</tr>
<tr>
<td>ora-lob-lb-locator</td>
<td>331, 1445</td>
</tr>
<tr>
<td>ora-lob-lb-locator-is-init</td>
<td>334, 1446</td>
</tr>
<tr>
<td>ora-lob-lb-open</td>
<td>335, 1446</td>
</tr>
<tr>
<td>ora-lob-read-buffer</td>
<td>333, 336, 1447</td>
</tr>
<tr>
<td>ora-lob-read-foreign-buffer</td>
<td>331, 333, 336, 1451</td>
</tr>
<tr>
<td>ora-lob-read-into-plain-file</td>
<td>336, 1450</td>
</tr>
<tr>
<td>ora-lob-svc-ctx-handle</td>
<td>331, 1452</td>
</tr>
<tr>
<td>ora-lob-trim</td>
<td>335, 1452</td>
</tr>
<tr>
<td>ora-lob-write-buffer</td>
<td>333, 336, 1453, 1456</td>
</tr>
<tr>
<td>ora-lob-write-foreign-buffer</td>
<td>331, 333, 336</td>
</tr>
<tr>
<td>ora-lob-write-from-plain-file</td>
<td>336</td>
</tr>
<tr>
<td>ora-lob-write-from-plain-file</td>
<td>1455</td>
</tr>
<tr>
<td>output</td>
<td>42</td>
</tr>
<tr>
<td>output-backtrace</td>
<td>654</td>
</tr>
<tr>
<td>output-stream-p</td>
<td>611</td>
</tr>
</tbody>
</table>

P
:p debugger command 15
package
  hiding 24
:package keyword 261
package-flagged-p function 1629
packages
  allocation of 133
*packages-for-warn-on-redefinition* variable 155, 844
parameters
  command line 392
  *default-character-element-type* 376, 378, 379, 380, 387, 964
parse-float function 845
parse-form-dspec function 696
parser generator main chapter 267
parser, error handling 270
passing runtime parameters 392
patches
  saving an image with 4
pathname comparison
Index

case-sensitivity on Mac OS X 407
pathname of deliverable 392, 1009
pathname of DLL 392, 1009
pathname of dynamic library 1009
pathname of executable 392, 1009
pathname of lisp image 392, 1009
pathname-location function 1019
pem-read function 352
pem-read function 524
pipe open 1625
pipe-exit-status function 1630
pipe-kill-process function 1631
PL/SQL 1399
platform 391
*features* 591
software-type 621
software-version 622
p-oci-env FLI type 332, 1457
p-oci-file FLI type 332, 1458
p-oci-lob-locator FLI type 332, 1458
p-oci-lob-or-file FLI type 1458
p-oci-svc-ctx FLI type 332, 1458
pointer-from-address function 1633
pointers weak 880
position-in-ring function 846
position-in-ring-forward function 846
PostgreSQL connecting 296
PostScript Printer Description files 157
PPD files 157–158
precompile-regexp function 1020
prepare-statement function 1459
:previous keyword 263
primitive-array-to-lisp-array function 1153
print-action-lists function 79, 1021
print-actions function 79, 1020
*print-binding-frames* variable 24, 655
*print-catch-frames* variable 24, 657
*print-command* variable 1022
printer configuring 157
*print-handler-frames* variable 24, 658
*print-invisible-frames* variable 660
*print-nickname* variable 1022
print-object generic function 157
*print-open-frames* variable 661
print-pretty-gesture-spec function 1634
print-profile-list function 142, 848
print-query function 299, 1461
*print-restart-frames* variable 24, 662
*print-string* variable 851
*print-symbols-using-bars* variable 1635
process creation 226
current 226
in LispWorks 225
scheduling 251
process exit status 401
:process keyword 42
process plist 249
process properties 249
process waiting 235
process-a-class-option generic function 433
process-alive-p function 1272
process-all-events function 1273
process-allow-scheduling function 226, 1274
process-arrest-reasons function 1274
process-a-slot-option generic function 435
process-break function 1275
process-continue function 1275
processes allocation of 134
processes-count function 227, 1316
process-exclusive-lock function 1276
process-exclusive-unlock function 1277
process-idle-time function 1278
*process-initial-bindings* variable 227, 1278
process-internal-server-p function 1279
process-interrupt function 245, 1280
process-interrupt-list
<table>
<thead>
<tr>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>process-join function</td>
<td>1281</td>
</tr>
<tr>
<td>process-kill function</td>
<td>1282</td>
</tr>
<tr>
<td>process-lock function</td>
<td>1283</td>
</tr>
<tr>
<td>process-mailbox function</td>
<td>1284</td>
</tr>
<tr>
<td>process-name function</td>
<td>1285</td>
</tr>
<tr>
<td>process-p function</td>
<td>1286</td>
</tr>
<tr>
<td>process-plist function</td>
<td>1287</td>
</tr>
<tr>
<td>process-poke function</td>
<td>1288</td>
</tr>
<tr>
<td>process-priority function</td>
<td>1289</td>
</tr>
<tr>
<td>process-private-property</td>
<td>1290</td>
</tr>
<tr>
<td>process-property</td>
<td>1291</td>
</tr>
<tr>
<td>process-reset function</td>
<td>1292</td>
</tr>
<tr>
<td>process-run-function</td>
<td>1293</td>
</tr>
<tr>
<td>process-run-reasons</td>
<td>1294</td>
</tr>
<tr>
<td>process-run-time function</td>
<td>1295</td>
</tr>
<tr>
<td>process-send function</td>
<td>1296</td>
</tr>
<tr>
<td>process-sharing-lock</td>
<td>1297</td>
</tr>
<tr>
<td>process-sharing-unlock</td>
<td>1298</td>
</tr>
<tr>
<td>process-stop function</td>
<td>1299</td>
</tr>
<tr>
<td>process-stopped function</td>
<td>1300</td>
</tr>
<tr>
<td>process-stopped-p function</td>
<td>1301</td>
</tr>
<tr>
<td>process-terminate function</td>
<td>1302</td>
</tr>
<tr>
<td>process-unlock function</td>
<td>1303</td>
</tr>
<tr>
<td>process-unstop function</td>
<td>1304</td>
</tr>
<tr>
<td>process-wait function</td>
<td>1305</td>
</tr>
<tr>
<td>process-wait-for-event</td>
<td>1306</td>
</tr>
<tr>
<td>process-wait-function</td>
<td>1307</td>
</tr>
<tr>
<td>process-wait-local function</td>
<td>1308</td>
</tr>
<tr>
<td>process-wait-local-with-periodic-checks function</td>
<td>1309</td>
</tr>
<tr>
<td>process-wait-local-with-timeout function</td>
<td>1310</td>
</tr>
<tr>
<td>process-wait-local-with-timeout-and-periodic-checks function</td>
<td>1311</td>
</tr>
<tr>
<td>process-wait-with-timeout function</td>
<td>1312</td>
</tr>
<tr>
<td>process-whostate function</td>
<td>1313</td>
</tr>
<tr>
<td>proclaim function</td>
<td>1314</td>
</tr>
<tr>
<td>product-registry-path</td>
<td>1315</td>
</tr>
<tr>
<td>profile macro</td>
<td>1316</td>
</tr>
<tr>
<td>profile time</td>
<td>1317</td>
</tr>
</tbody>
</table>

**profiler**
- interpretation of results 142
- main chapter 137
- pitfalls 142
- setting up 138
- *profiler-print-out-all* variable 850
- *profiler-threshold* variable 853
- profiler-tree-from-function function 854
- profiler-tree-to-function function 855
- *profile-symbol-list* variable 139, 854

**profiling**
- execution 137
- program 137
- program profiling 137
- promotion 120
- *prompt* variable 1022
- prompt in listener 1022
- *prompt* variable 8
- ps function 227, 1319
- pseudo operators
  - sql-boolean-operator 315
  - sql-function 315
  - sql-operator 315
- push macro 231
- pushnew-to-process-private-property function 250, 1317
- pushnew-to-process-property function 250, 1318

**Q**
- :q inspector command 27
- query function 304, 327, 1462
- *query-io* variable 623
- query-registry-key-info function 406, 1715
- query-registry-value function 406, 1716
- quick backtrace 14
- quit function 4, 401, 1023
- QuitLispWorks C function 163, 1773
- quitting LispWorks 4, 81
- :quit-when-no-windows delivery keyword 1033
Index

R
raw 32-bit arithmetic 410
raw 64-bit arithmetic 412
read-dhparams function 352, 526
read-eval-print loop 5, 1038
read-foreign-modules function 941
read-java-field function 1156
read-serial-port-char function 1355
read-serial-port-string function 1355
:read-timeout initarg 535
real time 139
rebinding macro 1025
reconnect function 292, 1464
record-definition function 696
record-java-class-lisp-symbol function 1158
record-message-in-windows-event-log function 1690
*record-source-files* variable 698
*redefinition-action* variable 698
redo 156
:redo listener command 6
reduce-memory function 133, 856
references-who function 858
regexp 996, 1026
registry API on Windows 406, 1705
registry-key-exists-p function 406, 1717
registry-value function 406, 1717
regular expression 996, 1026
regexp 996, 1026
regexp-find-symbols function 1026
relocate 396
remove-advice function 52, 58, 1027
remove-duplicates function 98
removef macro 1029
remove-from-process-private-property function 250, 1320
remove-from-process-property function 250, 1320
remove-process-private-property function 250, 1321
remove-process-property function 250, 1322
remove-special-free-action function 134, 858
remove-symbol-profiler function 139, 899
REPL 5, 1038
REPL inspector 25
replacement-source-form macro 699
replace-socket-stream-socket function 527
replace-standard-object function 437
report-error-to-java-host function 1159
:requires keyword 263, 264
*require-verbose* variable 1029
:res debugger command 21
reserved words 844
reset-code-coverage function 747
reset-code-coverage-snapshot function 814
reset-java-interface-for-new-jvm function 1160
reset-profiler function 139, 860
reset-ring function 861
restart 12
restart frame, examining 12
restart-case macro 614
:restarts keyword 662
restore-code-coverage-data function 747
restore-sql-reader-syntax-state function 317, 1465
:ret debugger command 21
reverse-subtract-code-coverage-data function 719
ring-length function 861
ring-name function 862
ringp function 861
ring-pop function 862
ring-push function 863
ring-ref function 864
rollback function 300, 302, 329, 1465
room function 127, 128, 131, 396, 614
room-values function 396, 1637
rotate-ring function 865
round-to-single-precision function 1030
:rules keyword 262
run-shell-command function 1638
runtime parameters 392
Index

S
: s inspector command 27
safe-locale-file-encoding function 1643
safety 87
save-argument-real-p function 866
save-code-coverage-data function 767
save-current-code-coverage function 767
save-current-session function 867
save-image function 147, 159, 392, 397, 401, 868
save-image-with-bundle function 877
save-tags-database function 700
save-universal-from-script function 878
saving images 147
sbchar accessor 378
sbchar function 1031
schar accessor 378
schedule-timer function 248, 1323
schedule-timer-milliseconds function 1325
schedule-timer-relative function 1327
schedule-timer-relative-milliseconds function 1328
scheduling of processes 251
security-description-string-for-open-named-pipe function 1692
segmentation violation in compiled code 90
select function 298, 309, 1466
select SQL operator 312
Self-contained examples
Asynchronous I/O 420
COMM package 419
DDE 420
miscellaneous examples 421
parser generator 420
save-image in a Mac OS X application bundle 420
socket streams 419
SSL 419
streams 420
TCP sockets 419
semaphore-acquire function 243,
semaphore-count function 243, 1331
semaphore-name function 243, 1332
semaphore-release function 243, 1332
semaphore-wait-count function 243, 1333
send-message-to-java-host function 1161
sequencep function 1031
serial-port function 1354
serial-port-input-available-p function 1356
server-terminate function 528
set-application-themed function 1696
set-approaching-memory-limit-callback function 1644
set-array-single-thread-p function 879
set-array-weak function 135, 880
set-automatic-gc-callback function 132, 1645
set-blocking-gen-num function 132, 1647
set-clos-initarg-checking function 439
set-code-coverage-snapshot function 814
setCurrentActivity Java method 1202
set-debugger-options function 663
set-default-character-element-type function 377, 380, 381, 1032
set-default-generation function 119, 128, 881
set-default-segment-size function 133, 1649
set-delay-promotion function 133, 1651
setErrorReporter Java method 1193
setf cdr-assoc function 1543
setf timer-name function 1335
set-file-dates function 1652
set-funcall-async-limit function 1334
set-gc-parameters function 121, 128, 882
set-gen-num-gc-threshold

1823
Index

setTextView Java method 1200
setup-atomic-funcall
function 1693
setup-field-accessor
function 1161
setup-java-caller function 1163
setup-java-constructor
function 1163
setup-lisp-proxy function 1164
set-up-profiler function 138, 892
set-verification-mode
function 529
*sg-default-size* variable 1664
:sh inspector command 27
shared libraries 159, 396
shared library 159
shared object file 159
Shift JIS 382
short-float type 418, 619
short-namestring function 392
short-site-name function 391, 620
Show Paths From Editor
command 98
showBugFormLogs Java method 1197
shutdown 81
sid-string-to-user-name
function 1698
simple-augmented-string
type 1535
simple-augmented-string-p
function 1535
simple-base-string type 377, 378,
563
simple-base-string-p
function 948
simple-bmp-string type 377, 950
simple-bmp-string-p function 951
simple-char type 1034
simple-char-p function 1034
simple-do-query macro 302, 328,
1471
SimpleInitLispWorks C
function 1772
simple-int32-vector type 411,
1665
simple-int64-vector type 412,
1665
simple-lock-and-condition-variable-wait
function 1334
simple-string type 377, 378
simple-text-string type 377, 378,
Index

special variables
*describe-level* 26
*directory-link-transparency* 585
speed 87
splash screen 1680
split-sequence function 1035
split-sequence-if function 1036
split-sequence-if-not function 1037
SQL
database functions 315
database operators 315
direct specification 304
mode 322
stored procedure 304, 1399
sql function 316, 1473
SQL pseudo operators
sql-boolean-operator 315, 1479
sql-function 315, 1478
sql-operator 315, 1478
sql-boolean-operator SQL pseudo operator 315, 1479
sql-connection-error SQL pseudo operator 315, 1479
sql-connection-error condition 1474
sql-connection-error-error 320
sql-database-data-error accessor 1475
sql-database-error accessor 1475
sql-error-database-message accessor 1475
sql-error-secondary-error-id accessor 1475
sql-expression function 316, 1476
sql-fatal-error condition 1477
sql-database-error error 320
*sql-enlarge-static* variable 1476
sql-error-database-message accessor 1475
sql-error-secondary-error-id accessor 1475
sql-expression function 316, 1476
sql-fatal-error condition 1477
sql-database-error error 320
sql-function pseudo operator 315
sql-function SQL pseudo operator 315, 1478
*sql-libraries* variable 291, 1477
*sql-loading-verbose* variable 291, 301, 1478
sql-operation function 315, 1478

simple-text-string-p
function 1043
single-float type 418, 621
single-form-form-parser function 701
single-form-with-options-form-parser function 702
single-threaded arrays 232
: size initarg 1668
:sjis external format 382
SLIME 3
slot-boundp-using-class generic function 213, 440
slot-makunbound-using-class generic function 213, 441
slot-value atomic operations 254
slot-value-using-class generic function 213, 442
: socket initarg 535
socket-error class 535
socket-stream class 351, 356, 535
socket-stream-address function 541
socket-stream-ctx function 351, 541
socket-stream-peer-address function 542
socket-stream-shutdown function 543
socket-stream-socket accessor 536
socket-stream-ssl function 351, 544
software-type function 391, 621
software-version function 391, 622
some SQL operator 312
sort-inspector-p generic function 999
source level debugging 70, 98, 905
source-debugging-on-p function 896
*source-found-action* variable 153
: source-only keyword 262
space 87
special actions 134
special forms declare 86, 91, 573

1042
simple-text-string-p
function 1043
single-float type 418, 621
single-form-form-parser function 701
single-form-with-options-form-parser function 702
single-threaded arrays 232
: size initarg 1668
:sjis external format 382
SLIME 3
slot-boundp-using-class generic function 213, 440
slot-makunbound-using-class generic function 213, 441
slot-value atomic operations 254
slot-value-using-class generic function 213, 442
: socket initarg 535
socket-error class 535
socket-stream class 351, 356, 535
socket-stream-address function 541
socket-stream-ctx function 351, 541
socket-stream-peer-address function 542
socket-stream-shutdown function 543
socket-stream-socket accessor 536
socket-stream-ssl function 351, 544
software-type function 391, 621
software-version function 391, 622
some SQL operator 312
sort-inspector-p generic function 999
source level debugging 70, 98, 905
source-debugging-on-p function 896
*source-found-action* variable 153
: source-only keyword 262
space 87
special actions 134
special forms declare 86, 91, 573
Index

sql-operator function 316, 1480
sql-operator pseudo operator 315
sql-operator SQL pseudo operator 315, 1478
sql-recording-p function 319, 1481
sql-stream function 319, 1482
sql-temporary-error condition 1483
sql-temporary-error error 320
sql-timeout-error condition 1483
sql-timeout-error error 320
sql-user-error condition 1483
sql-user-error error 320
square bracket syntax 311
ssl-add-client-ca function 353
ssl-add-client-ca function 353
ssl-cipher-get-bits function 353
ssl-cipher-get-bits function 353
ssl-cipher-get-name function 353
ssl-cipher-get-name function 353
ssl-cipher-get-version function 353
ssl-cipher-pointer FLI type 353, 544
ssl-cipher-pointer-stack FLI type 353, 544
ssl-clear-num-renegotiations function 353
ssl-closed class 361, 545
ssl-condition class 361, 545
:ssl-configure-callback initarg 536
:ssl-ctrl function 353
:ssl-ctx initarg 535
ssl-ctx-add-client-ca function 353
ssl-ctx-add-client-ca function 353
ssl-ctx-add-extra-chain-cert function 353
ssl-ctx-add-extra-chain-cert function 353
ssl-ctx-get-max-cert-list function 353
ssl-ctx-get-mode function 353
ssl-ctx-get-options function 354
ssl-ctx-get-read-ahead function 354
ssl-ctx-get-verify-mode function 354
ssl-ctx-load-verify-locations function 354
ssl-ctx-get-temp-rsa function 354
ssl-ctx-pointer FLI type 353, 546
ssl-ctx-sess-get-cache-mode function 354
ssl-ctx-sess-get-cache-size function 354
ssl-ctx-sess-set-cache-mode function 354
ssl-ctx-sess-set-cache-size function 354
ssl-ctx-set-client-ca-list function 354
ssl-ctx-set-max-cert-list function 354
ssl-ctx-set-mode function 354
ssl-ctx-set-options function 354
ssl-ctx-set-read-ahead function 354
ssl-ctx-set-temp-dh function 354
ssl-ctx-set-temp-rsa function 354
ssl-ctx-use-certificate-file function 354
ssl-ctx-use-certificate-file function 354
ssl-ctx-use-certificate-file function 354
ssl-ctx-use-privatekey-file function 354
ssl-ctx-use-rsaprivatekey-file function 354
ssl-error class 361, 546
ssl-error class 361, 546
ssl-get-current-cipher function 354
ssl-get-current-cipher function 354
ssl-get-max-cert-list function 354
ssl-get-mode function 355
ssl-get-options function 355
ssl-get-verify-mode function 355
ssl-get-version function 355
ssl-load-client-ca-file function 355
ssl-load-client-ca-file function 355
ssl-new function 359, 547
ssl-num-renegotiations function 355
ssl-pointer FLI type 353, 547
ssl-session-reused function 355
ssl-set-accept-state function 355, 358
ssl-set-accept-state function 355, 358
ssl-set-client-ca-list function 355
ssl-set-connect-state function 355, 358
ssl-set-connect-state function 355, 358
ssl-set-max-cert-list function 355
ssl-set-mode function 355
ssl-set-options function 355
ssl-set-temp-dh function 355
ssl-set-temp-dh function 355
Index

ssl-set-tmp-rsa function 355
:ssl-side initarg 535
ssl-total-renegotiations function 355
ssl-use-certificate-file function 355
ssl-use-privatekey-file function 355
ssl-use-rsaprivatekey-file function 355
ssl-x509-lookup class 361, 548
stack examining 12
extension 799
stack size 134, 775, 1550, 1664
*stack-overflow-behaviour* variable 1666
stacks allocation of 134
standard-accessor-method class 214
standard-class class 215
standard-db-object class 305, 1484
*standard-input* variable 623
standard-instance-access function 214
standard-java-object class 1165
standard-object class 230
*standard-output* variable 623
standard-reader-method class 214
standard-writer-method class 214
start 81
start LispWorks 1
start-dde-server function 281, 1765
start-gc-timing function 897
starting LispWorks 1, 81
start-profiling function 139, 140, 900
start-sql-recording function 319, 1486
storage management main chapter 111
storage-exhausted class 1668
storage-exhausted-gen-num accessor 1668
storage-exhausted-size accessor 1668
storage-exhausted-static accessor 1668
storage-exhausted-type accessor 1668
str FLI type 1698
:stream initarg 535
stream-advance-to-column generic function 1503
stream-check-eof-no-hang generic function 1504
stream-clear-input generic function 342, 1504
stream-clear-output generic function 343, 1505
stream-element-type generic function 340, 627
stream-file-position generic function 1506
stream-fill-buffer generic function 1506
stream-finish-output generic function 343, 1507
stream-flush-buffer generic function 1508
stream-force-output generic function 343, 1509
static object allocation in memory management 119
staticp function 1667
status function 292, 1485
status Java method 1189
stchar accessor 378
stchar function 1039
stderr 1614
:step keyword 40
step macro 624
*step-compiled* variable 627
*step-filter* variable 627
stepper, entering when tracing 40
*step-print-env* variable 627
stop-gc-timing function 897
stop-profiling function 139, 140, 900
stop-sql-recording function 319, 1486
storage management main chapter 111
storage-exhausted class 1668
storage-exhausted-gen-num accessor 1668
storage-exhausted-size accessor 1668
storage-exhausted-static accessor 1668
storage-exhausted-type accessor 1668
str FLI type 1698
:stream initarg 535
stream-advance-to-column generic function 1503
stream-check-eof-no-hang generic function 1504
stream-clear-input generic function 342, 1504
stream-clear-output generic function 343, 1505
stream-element-type generic function 340, 627
stream-file-position generic function 1506
stream-fill-buffer generic function 1506
stream-finish-output generic function 343, 1507
stream-flush-buffer generic function 1508
stream-force-output generic function 343, 1509

1827
stream-fresh-line generic function 1509
stream-line-column generic function 343, 1510
stream-listen generic function 342, 1511
stream-output-width generic function 1512
stream-peek-char generic function 1512
stream-read-buffer generic function 1513
stream-read-byte generic function 1514
stream-read-char generic function 341, 1515
stream-read-char-no-hang generic function 1515
stream-read-line generic function 1516
stream-read-sequence generic function 1517
stream-read-timeout accessor 536
stream-read-timeout generic function 1518
streams
  defining new 340
directionality 340
example 340-345
input 341
instantiating 344
output 342
user defined 339
stream-start-line-p generic function 343, 1518
stream-terpri generic function 1519
stream-unread-char generic function 341, 1520
stream-write-buffer generic function 1520
stream-write-byte generic function 1521
stream-write-char generic function 342, 1522
stream-write-sequence generic function 1522
stream-write-string generic function 1523
stream-write-timeout accessor 536
string construction 379
string type 376, 377, 628
string types 376
string-append function 1039
string-append* function 1040
string-ip-address function 554
structure types
  async-io-state 459
structurep function 1041
subtract-code-coverage-data function 719
superclass
  invalid 217
  sweep 120
sweep-all-objects function 134, 901
sweep-gen-num-objects function 1668
switch-open-tcp-stream-with-ssl-to-java function 555
switch-static-allocation function 118, 119, 902
symbol macros
  +int32-0+ 1583
  +int32-1+ 1583
  +int64-0+ 1594
  +int64-1+ 1594
*symbol-alloc-gen-num* variable 128, 133, 903
symbol-dynamically-bound-p function 903
symbolic query syntax 310
symbolic syntax in Common SQL 310
symbols
  allocation of 133
symeval-in-process function 251, 1335
Synchronization barriers 242
syntax, in Common SQL 310
system
  compile 954
  defining 260–265
  introduction to 259–265
  load 1013
  members of 261
  plan 262
  print 1007
  rules 262–264
system commands
  running directly 1538
  running via a shell 1538
T
table-exists-p function 1487
Index

tail call 95
tail call merging 95
tail call optimization 95
tail merge 95
tail recursion 95
tail-call 95
TCP/IP socket
  client side 348
  server side 347
SSL interface 351
teletype inspector 25
temp files 405
temporary files 405
*terminal-debugger-block-multiprocessing*
  variable 664
text-string type 376, 1042
text-string-p function 1043
threads 225
  allocation of 134
throw-an-exception function 1167
throw-if-tag-found macro 904
time macro 629
timer-expired-p function 1336
timer-name function 1337
timers 248
  I/O 249
  input and output 249
  multiprocessing 249
  process 249
  threading issues 249
toggle-source-debugging
  function 99, 905
tools
  inspector 25
:top debugger command 21
top-level loop 5
total-allocation function 127, 906
trace
  excessive output 47
  main chapter 35
  missing output 47
  not working 47
  troubleshooting 47
trace macro 631
traceable-dspec-p function 703
traced functions
  arguments 38
*traced-arglist* variable 38, 47, 907
*traced-results* variable 38, 47, 632, 908
*trace-indent-width* variable 46, 909
*trace-level* variable 46, 910
trace-new-instances-on-access function 443
trace-on-access function 444
:trace-output keyword 42
*trace-output* variable 46, 623
*trace-print-circle* variable 47, 911
*trace-print-length* variable 26, 47, 911
*trace-print-level* variable 26, 47, 912
*trace-print-pretty* variable 47, 913
tracer
  :after option 38
  :allocation option 43
  :before option 37
  :break option 39
  :break-on-exit option 39
  commands available 37–44
  definition specs 46
  directing output 42
  entering the stepper 40
  :entrycond option 40
  :eval-after option 39
  :eval-before option 38
  evaluating forms 37–39
  example of use 35
  :exitcond option 40
  functions, tracing inside 43
  information displayed 36
  :inside option 43
  invoking the debugger 39
  memory allocation 43
  methods, tracing 45
  :process option 42
  restricting to a process 42
  :step option 40
  traced function, arguments for 38
  traced functions, results for 38
  :trace-output option 42
*trace-verbose* variable 914
tracing functions
  inside other functions 43
tracing-enabled-p function 703
tracing-state function 704
transaction handling
  in Common SQL 293, 301, 324
true function 1043
Index

true-name function 638
try-compact-in-generation function 126, 128, 915
try-move-in-generation function 126, 128, 917
tstr FLI type 1699
tty 869
:type init arg 1668
typed-aref function 1669
types
16-bit-string 944
8-bit-string 943
accepting-handle 454
augmented-string 1535
base-char 374, 946
base-character 946
base-string 376, 563
bmp-char 374, 948
bmp-string 376, 950
character 374
code-coverage-data 748
code-coverage-file-stats 757
double-float 590
extended-char 992
extended-character 992
fixnum 415
int32 410, 1581
int64 412, 1592
java-vm-poi 1133
jboolean 1134
jbyte 1134
jchar 1134
jdouble 1134
jfloat 1134
jint 1134
jlong 1134
jni-env-poi 1134
jobject 1135
jshort 1134
long-float 598
mt-random-state 1018
short-float 418, 619
simple-augmented-string 1535
simple-base-string 377, 378, 563
simple-bmp-string 377, 950
simple-char 1034
simple-int32-vector 411, 1665
simple-int64-vector 412, 1665
simple-string 377, 378
simple-text-string 377, 378, 1042
single-float 418, 621
string 376, 377, 628
text-string 376, 1042
unlocked-queue 918

U
:u inspector command 27
:ud inspector command 27, 28
unbreak-new-instances-on-access function 447
unbreak-on-access function 447
undefined-action macro 78, 1044
undefined-action-list macro 78, 1045
Unicode 373
:unicode external format 382
unicode-alpha-char-p function 389, 1045
unicode-alphanumericp function 389, 1046
unicode-both-case-p function 389, 1047
unicode-char-equal function 388, 1048
unicode-char-greaterp function 388, 1049
unicode-char-lesserp function 388, 1049
unicode-char-not-equal function 388, 1048
unicode-char-not-greaterp function 388, 1050
unicode-char-not-lesspp function 388, 1050
unicode-lower-case-p function 389, 1051
unicode-lower-case-p function 389, 1051
unicode-string-equal function 389, 1053
unicode-string-greaterp function 389, 1053
unicode-string-lesserp function 389, 1053
unicode-string-not-equal function 389, 1051
unicode-string-not-greaterp function 389, 1054
unicode-string-not-lesspp function 389, 1054
unicode-upper-case-p function 389, 1055
union SQL operator 312
universal binaries
  helper functions 740, 866
saving
  advanced 775
  simply 878
UNIX command
  call-system 1537
  call-system-showing-output 1540
  open-pipe 1625
  run-shell-command 1640
Unix commands
  calling from Lisp 400
Unix functions
  calling from Lisp 400
unixODBC 337
unlocked-queue type 918
unlocked-queue-read function 836
unlocked-queue-ready function 836
unlocked-queue-send function 836
unnotice-fd function 1339
*unresolved-messages* 942
unschedule-timer function 1339
untrace macro 639
untrace-new-instances-on-access function 448
untrace-on-access function 448
unwind-protect-blocking-interrupts macro 245, 919
unwind-protect-blocking-interrupts-in-cleanups macro 245, 920
update-instance-for-different-class function 640
update-instance-for-redefined-class function 641
update-instance-from-records generic function 309, 1487
update-objects-joins function 1488
update-record-from-slot generic function 309, 1491
update-records function 300, 301, 328, 1490
update-records-from-instance generic function 309, 1491
update-slot-from-record generic function 309, 1492
URL
  opening 1629
:use listener command 7
user defined stream 339
user-homedir-pathname
  function 193, 404, 1577
user-name-to-sid-string function 1701
user-preference function 402, 1056
UTF-16 382
:utf-16 external format 382
:utf-16be external format 384, 710
:utf-16le external format 384, 710
:utf-16-native external format 384, 710
:utf-16-reversed external format 384, 710
:utf-32 external format 382
UTF-8 382, 386
:utf-8 external format 382
utilities in Common SQL 317
V
:V debugger command 16
validate-superclass generic function 215, 216
valid-external-format-p function 718
variable-information function 921
variables
$ (dollar) 26
$$ 26
$$$ 26
*active-finders* 671
*autoload-asdf-integration* 945
*background-input* 736
*background-output* 736
*background-query-io* 736
*binary-file-type* 571, 1536
*binary-file-types* 571, 1536
*browser-location* 951
*cache-table-queries-default* 1365
*check-network-server* 1544
*compiler-break-on-error* 765
*connect-if-exists* 1373
*current-process* 226, 1224
*debug-initialization-errors-in-snap-shot* 1549
*debug-io* 23, 623
*debug-print-length* 23, 647
*debug-print-level* 24, 648
*default-action-list-sort-time* 79, 964
Index

*default-database* 1382
*default-database-type* 1382
*default-libraries* 941
*default-package-use-list* 777
*default-process-priority* 1233
*default-profiler-collapse* 778
*default-profiler-limit* 779
*default-profiler-sort* 779
*default-stack-group-list-length* 134, 1550
*default-update-objects-max-len* 1383
*defstruct-generates-print-object-method* 157
*defsystem-verbose* 975
*describe-length* 27, 977
*describe-level* 978
*describe-print-length* 26, 980
*describe-print-level* 26, 980
*directory-link-transparency* 1558
*disable-trace* 788
*dspec-classes* 684
*enter-debugger-directly* 985
*error-output* 623
*extended-spaces* 1060, 1561
*external-formats* 993
*features* 591
*file-encoding-detection-algorithm* 386, 1562
*file-eol-style-detection-algorithm* 386, 1564
*filename-pattern-encoding-matches* 1564
gesture-spec-accelerator-bit 1568
gesture-spec-caps-lock-bit 1568
gesture-spec-control-bit 1568
gesture-spec-hyper-bit 1568
gesture-spec-meta-bit 1568
gesture-spec-shift-bit 1568
gesture-spec-super-bit 1568
*grep-command* 1002
*grep-command-format* 1003
*grep-fixed-args* 1004
*handle-existing-action-in-action-list* 78, 1004
*handle-existing-action-list* 78, 1005
*handle-existing-defpackage* 827
*handle-missing-action-in-action-list* 79, 1006
*handle-missing-action-list* 79, 1005
*handle-old-in-package* 828
*handle-old-in-package-used-as-make-package* 829
*handle-warn-on-redefinition* 155, 1006
*hidden-packages* 24, 650
*init-file-name* 1008
*initialized-database-types* 1402
*initial-processes* 163, 227, 1242
*inspect-print-length* 27
*inspect-print-level* 27
*inspect-through-gui* 1009
*latin-1-code-pages* 1684
*line-arguments-list* 392, 1604
*lispworks-directory* 1010
*load-fasl-or-lisp-file* 833
*main-process* 1261
*maximum-ordinary-windows* 154
*mt-random-state* 1017
*multibyte-code-page-eff* 337, 1686
*mysql-library-directories* 295, 296, 1415
*mysql-library-path* 295, 296, 1416
*mysql-library-subdirectories* 295, 1417
*packages-for-warn-on-redefinition* 155, 844
*print-binding-frames* 24, 655
*print-catch-frames* 24, 657
*print-command* 1022
*print-handler-frames* 24,
Index

*print-invisible-frames* 660
*print-nickname* 1022
*print-open-frames* 661
*print-restart-frames* 24, 662
*print-string* 851
*print-symbols-using-bars* 1635
*process-initial-bindings* 227, 1278
*profiler-print-out-all* 850
*profiler-threshold* 853
*profile-symbol-list* 139, 854
*prompt* 8, 1022
*query-io* 623
*record-source-files* 698
*redefinition-action* 155, 698
*require-verbose* 1029
*sg-default-size* 1664
*source-found-action* 153
*sql-enlarge-static* 1476
*sql-libraries* 291, 1477
*sql-loading-verbose* 291, 301, 1478
*stack-overflow-behaviour* 1666
*standard-input* 623
*standard-output* 623
*step-compiled* 627
*step-filter* 627
*step-print-env* 627
*symbol-alloc-gen-num* 128, 133, 903
*terminal-debugger-block-multiprocessing* 664
*traced-arglist* 38, 47, 907
*traced-results* 38, 47, 632, 908
*trace-indent-width* 46, 909
*trace-level* 46, 910
*trace-output* 46, 623
*trace-print-circle* 47, 911
*trace-print-length* 26, 47, 911
*trace-print-level* 26, 47, 912
*trace-print-pretty* 47, 913
*trace-verbose* 914
vector-pop function 230, 232
vector-push function 230, 232
vector-push-extend function 230, 232
verbose backtrace 14

verify-java-caller function 1168
verify-java-callers function 1169
verify-lisp-proxies function 1172
verify-lisp-proxy function 1172
virtual (ordinary) slots 307
virtual time 139

W
wait-for-connection generic function 1701
wait-for-input-streams function 1671
wait-for-input-streams-returning-first function 1672
wait-for-wait-state-collection function 556
wait-processing-events function 1340
wait-serial-port-state function 1357
wait-state-collection class 557
wait-state-collection-stop-loop function 557
weak
arrays 880
hash tables 602
weak hash tables 602
weak pointers 880
web browser 1629
:when keyword 43
when-let macro 1058
when-let* macro 1059
whitespace-char-p function 1060
who-binds function 923
who-calls function 924
who-references function 924
who-sets function 925
Windows code page 936 382
Windows event log 1690
Windows registry
API 406, 1705
Windows XP themes 1696
windows-cp936 382
:windows-cp936 external format 382
with-action-item-error-handling macro 1061
with-action-list-mapping macro 1062
with-code-coverage-genera-
Index

:write-timeout initarg 535

wstr FLI type 1700

X
xrefs 738, 741, 905, 923, 924

Y
yellow pages 502

yield function 1347

tion macro 926

with-dde-conversation macro 277, 1780

with-debugger-wrapper macro 23, 666

with-ensuring-gethash macro 926

with-exclusive-lock macro 233, 1342

with-hash-table-locked macro 230, 231, 928

with-heavy-allocation macro 128, 929

with-interrupts-blocked macro 245, 1343

with-lock macro 233, 1343

with-modification-change macro 1673

with-modification-check macro macro 1674

with-noticed-socket-stream macro 588

with-other-threads-disabled macro 245, 1675

without-code-coverage macro 935

without-interrupts macro 245, 1345

without-preemption macro 245, 1346

with-output-to-fasl-file macro 930

with-output-to-string macro 641

with-registry-key macro 406, 1720

with-ring-locked macro 934

with-sharing-lock macro 233, 1344

with-stream-input-buffer macro 1524

with-stream-output-buffer macro 1526

with-transaction macro 300, 301, 1493

with-unique-names macro 1063

with-windows-event-log-event-source macro 1702

write-java-class-definitions-to-file function 1174

write-java-class-definitions-to-stream function 1174

write-serial-port-char function 1358

write-serial-port-string