Contents

Preface xxxv

1 Starting LispWorks 1
   The usual way to start LispWorks 1
   Passing arguments to LispWorks 1
   Starting the Common 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 control variables 22

4 The REPL Inspector 25
   Describe 25
   Inspect 26
Contents

Inspection modes 28

5 The Trace Facility 35

Simple tracing 35
Tracing options 37
Example 44
Tracing methods 45
Trace variables 45

6 The Advice Facility 49

Defining advice 49
Combining the advice 51
Removing advice 53
Advice for macros and methods 53
Examples 56
Advice functions and macros 59

7 Dspecs: Tools for Handling Definitions 61

Dspecs 61
Forms of dspecs 62
Dspec namespaces 63
Types of relations between definitions 66
Details of system dspec classes and aliases 68
Subfunction dspecs 70
Tracking definitions 71
Finding locations 72
Users of location information 73

8 Action Lists 77

Defining and undefining action lists 78
Exception handling variables 80
Other variables 81
Diagnostic utilities 81
Examples 82
Standard Action Lists 84

9 The Compiler 85
10 Storage Management 103

Introduction 103
Generations and segments 104
Memory Management in 32-bit LispWorks 104
Memory Management in 64-bit LispWorks 112
Common Memory Management Features 116
Assisting the Garbage Collector 118

11 The Profiler 121

What the profiler does 121
Setting up the profiler 122
Running the profiler 123
Profiler output 125
Interpretation of profiling results 126
Profiling pitfalls 126
Profiling and garbage collection 127

12 Customization of LispWorks 129

Introduction 129
Configuration and initialization files 130
Saving a LispWorks image 131
Saved sessions 133
Load and open your files on startup 136
Customizing the editor 136
Finding source code 138
Controlling redefinition warnings 138
Specifying the initial working directory 139
Using ! for :redo 139
Customizing LispWorks for use with your own code 139
Contents

Structure printing 140
Configuring the printer 140

13 LispWorks as a dynamic library 143
Introduction 143
Creating a dynamic library 144
Initialization of the dynamic library 145
Relocation 146
Multiprocessing in a dynamic library 147
Unloading a dynamic library 147

14 The Metaobject Protocol 149
Metaobject features incompatible with AMOP 149
Common problems when using the MOP 152
Implementation of virtual slots 153

15 Multiprocessing 161
Introduction to processes 161
The process programming interface 162
Atomicity and thread safety of the LispWorks implementation 172
Low level atomic operations 175
Aids for implementing modification checks 176
Ensuring order of memory between operations in different threads 177
Locks 179
Process Waiting 182
Synchronization between threads 186
Timers 189
Process properties 189
Native threads and foreign code 190
Example 191

16 Common Defsystem 193
Introduction 193
Defining a system 194

17 The Parser Generator 201
Introduction 201
Grammar rules 201
Functions defined by defparser 203
Error handling 204
Interface to lexical analyzer 204
Example 205

18 Dynamic Data Exchange 209
Introduction 209
Client interface 211
Server interface 214

19 Common SQL 219
Introduction 219
Initialization 223
Functional interface 230
Object oriented interface 238
Symbolic SQL syntax 243
Working with date fields 249
SQL I/O recording 251
Error handling in Common SQL 251
Using MySQL 253
Using Oracle 258
Oracle LOB interface 259

20 User Defined Streams 269
Introduction 269
An illustrative example of user defined streams 269

21 Socket Stream SSL interface 277
Creating a stream with SSL 277
SSL-CTX and SSL objects 278
OpenSSL interface 278
Socket Stream SSL keyword arguments 282
Attaching SSL to an existing socket-stream 285
Using SSL objects directly 285
Initialization 286
Obtaining and installing the OpenSSL library 286
## Errors in SSL

22 **Internationalization** 289  
Introduction 289  
Character and String types 289  
String accessors 292  
String Construction 292  
External Formats 295  
External Formats and File Streams 296  
External Formats and the Foreign Language Interface 299  
Unicode character and string functions 299

## LispWorks' Operating Environment

23 The Operating System 301  
Site Name 301  
The Lisp Image 302  
The Command Line 302  
Address Space and Image Size 305  
Startup relocation 306  
Calling external programs 311  
Snapshot debugging of startup errors 311  
System message log 312  
Exit status 312  
Creating a new executable with code preloaded 312  
Universal binaries on Mac OS X 312  
User Preferences 313  
Accessing the Windows registry 314  
The home directory 315  
Special Folders 316

## 64-bit LispWorks

24 Introduction 319  
Heap size 320  
Architectural constants 320  
Speed 321  
Memory Management 321  
Float types 321
External libraries 321

25 The CLOS Package 323

break-new-instances-on-access 323
break-on-access 324
class-extra-initargs 325
compute-class-potential-initargs 326
compute-discriminating-function 328
funcallable-standard-object 328
process-a-class-option 329
process-a-slot-option 331
set-make-instance-argument-checking 334
slot-boundp-using-class 335
slot-makunbound-using-class 336
slot-value-using-class 336
trace-new-instances-on-access 337
trace-on-access 338
unbreak-new-instances-on-access 341
unbreak-on-access 342
untrace-new-instances-on-access 342
untrace-on-access 343

26 The COMM Package 345

attach-ssl 345
destroy-ssl 347
destroy-ssl-ctx 347
detach-ssl 348
do-rand-seed 349
ensure-ssl 349
get-host-entry 350
get-socket-address 352
get-socket-peer-address 352
get-verification-mode 353
ip-address-string 354
make-ssl-ctx 354
open-tcp-stream 355
openssl-version 358
pem-read 359
read-dhparams 360
Contents

x

set-verification-mode 362
set-ssl-ctx-dh 364
set-ssl-ctx-options 365
set-ssl-ctx-password-callback 366
set-ssl-library-path 367
socket-error 367
socket-stream 368
socket-stream-address 373
socket-stream-ctx 373
socket-stream-peer-address 374
socket-stream-ssl 374
ssl-cipher-pointer 375
ssl-cipher-pointer-stack 375
ssl-closed 376
ssl-condition 376
ssl-ctx-pointer 376
ssl-error 377
ssl-failure 377
ssl-new 377
ssl-pointer 378
ssl-x509-lookup 378
start-up-server 379
start-up-server-and-mp 383
string-ip-address 384
with-noticed-socket-stream 385

27 The COMMON-LISP Package 387

apropos 387
apropos-list 388
base-string 389
close 389
coerce 390
compile 391
compile-file 392
concatenate 397
declaim 397
declare 398
defclass 403
defpackage 407
describe 409
directory 409
disassemble 414
documentation 415
double-float 416
*features* 416
input-stream-p 419
interactive-stream-p 420
load-logical-pathname-translations 421
long-float 421
long-site-name 422
loop 422
make-array 424
make-hash-table 425
make-instance 428
make-sequence 429
map 429
merge 430
open 431
open-stream-p 433
output-stream-p 433
proclaim 434
restart-case 436
room 436
short-float 440
short-site-name 441
simple-base-string 441
single-float 441
software-type 442
software-version 443
step 444
stream-element-type 447
string 448
time 449
trace 451
truename 458
untrace 459
update-instance-for-different-class 460
update-instance-for-redefined-class 460
with-output-to-string 461
Contents

28 The COMPILER Package 463
    deftransform 463

29 The DBG Package 467
    *debug-print-length* 467
    *debug-print-level* 468
    executable-log-file 469
    *hidden-packages* 470
    log-bug-form 471
    logs-directory 473
    output-backtrace 474
    *print-binding-frames* 475
    *print-catch-frames* 477
    *print-handler-frames* 478
    *print-open-frames* 479
    *print-restart-frames* 480
    *terminal-debugger-block-multiprocessing* 481
    with-debugger-wrapper 484

30 The DSPEC Package 489
    *active-finders* 489
    at-location 490
    canonicalize-dspec 491
    def 492
    define-dspec-alias 493
    define-dspec-class 494
    define-form-parser 497
    dspec-class 501
    *dspec-classes* 501
    dspec-defined-p 502
    dspec-definition-locations 502
    dspec-equal 503
    dspec-name 504
    dspec-primary-name 504
    dspec-progenitor 505
    dspec-subclass-p 506
    dspec-undefined 506
    discard-source-info 507
find-dspec-locations 507
find-name-locations 508
get-form-parser 509
local-dspec-p 510
location 511
name-defined-dspecs 511
name-definition-locations 512
name-only-form-parser 513
parse-form-dspec 514
record-definition 514
*record-source-files* 516
*redefinition-action* 516
save-tags-database 517
single-form-form-parser 517
single-form-with-options-form-parser 518
traceable-dspec-p 519
tracing-enabled-p 520
tracing-state 521

31 The EXTERNAL-FORMAT Package 523
char-external-code 523
decode-external-string 524
encode-lisp-string 525
external-format-error 526
external-format-foreign-type 526
external-format-type 527
find-external-char 527
valid-external-format-p 528

32 The HCL Package 531
add-special-free-action 531
add-symbol-profiler 532
allocation-in-gen-num 533
analysing-special-variables-usage 534
array-weak-p 537
avoid-gc 538
binds-who 539
block-promotion 539
building-universal-intermediate-p 541
calls-who 541

- cd 542
- change-directory 543
- check-fragmentation 543
- clean-down 544
- clean-generation-0 546
- collect-generation-2 547
- collect-highest-generation 547
- *compiler-break-on-error* 548
- compile-file-if-needed 549
- copy-to-weak-simple-vector 550
- create-macos-application-bundle 551
- create-universal-binary 553
- current-stack-length 555
- *default-package-use-list* 555
- *default-profiler-collapse* 556
- *default-profiler-cutoff* 556
- *default-profiler-limit* 557
- *default-profiler-sort* 557
- defglobal-parameter 558
- defglobal-variable 558
- delete-advice 559
- *disable-trace* 560
- do-profiling 561
- dump-form 563
- dump-forms-to-file 564
- enlarge-generation 565
- enlarge-static 566
- expand-generation-1 567
- extend-current-stack 568
- extended-time 569
- file-string 571
- file-writable-p 572
- find-object-size 572
- finish-heavy-allocation 573
- flag-not-special-free-action 574
- flag-special-free-action 575
- gc-generation 575
- gc-if-needed 579
- get-default-generation 579
source-debugging-on-p 633
start-profiling 634
stop-profiling 636
sweep-all-objects 637
switch-static-allocation 638
*symbol-alloc-gen-num* 638
toggle-source-debugging 639
total-allocation 640
*traced-arglist* 641
*traced-results* 642
*trace-indent-width* 643
*trace-level* 644
*trace-print-circle* 645
*trace-print-length* 646
*trace-print-level* 647
*trace-print-pretty* 648
*trace-verbose* 649
try-compact-in-generation 650
try-move-in-generation 651
unwind-protect-blocking-interrupts 653
unwind-protect-blocking-interrupts-in-cleanups 654
who-binds 655
who-calls 656
who-references 657
who-sets 657
with-hash-table-locked 658
with-heavy-allocation 659
with-output-to-fasl-file 659

33 The LINK-LOAD Package 663
break-on-unresolved-functions 663
foreign-symbol-address 664
get-foreign-symbol 665
lisp-name-to-foreign-name 666
read-foreign-modules 667

34 The LISPWORKS Package 669
8-bit-string 669
16-bit-string 670
appendf 670
base-character 670
base-character-p 671
base-char-p 671
base-char-code-limit 672
base-string-p 672
*browser-location* 673
call-next-advice 673
choose-unicode-string-hash-function 675
compile-system 675
concatenate-system 677
current-pathname 679
defadvice 681
*default-action-list-sort-time* 684
*default-character-element-type* 684
define-action 685
define-action-list 687
defsystem 689
*defsystem-verbose* 694
delete-directory 694
deliver 695
*describe-length* 696
*describe-level* 696
*describe-print-length* 698
*describe-print-level* 698
dll-quit 699
dotted-list-length 701
dotted-list-p 702
do-nothing 702
*enter-debugger-directly* 703
environment-variable 703
erro-value 705
example-file 705
example-compile-file 706
example-load-binary-file 707
execute-actions 707
extended-char 709
extended-character 709
extended-character-p 710
extended-char-p 710
*external-formats* 711
false 712
file-directory-p 712
find-regexp-in-string 713
function-lambda-list 715
get-inspector-values 716
get-unix-error 718
*grep-command* 718
*grep-command-format* 719
*grep-fixed-args* 720
*handle-existing-action-in-action-list* 720
*handle-existing-action-list* 721
*handle-missing-action-list* 721
*handle-missing-action-in-action-list* 722
*handle-warn-on-redefinition* 722
hardcopy-system 723
*init-file-name* 724
*inspect-through-gui* 724
lisp-image-name 725
*lispworks-directory* 726
load-all-patches 727
load-system 728
make-unregistered-action-list 730
make-mt-random-state 731
mt-random 732
*mt-random-state* 733
mt-random-state 733
mt-random-state-p 734
pathname-location 734
precompile-regexp 735
print-actions 736
print-action-lists 736
*print-command* 737
*print-nickname* 737
*prompt* 738
quit 739
rebinding 740
regexp-find-symbols 741
remove-advice 742
removef 744
*require-verbose* 744
round-to-single-precision 745
sbchar 746
set-default-character-element-type 746
simple-base-string-p 747
simple-char 748
simple-char-p 748
simple-text-string 749
simple-text-string-p 749
split-sequence 750
split-sequence-if 751
split-sequence-if-not 752
start-tty-listener 753
stchar 753
string-append 754
text-string 755
text-string-p 756
true 756
undefine-action 757
undefine-action-list 757
unicode-alpha-char-p 758
unicode-alphanumeric-p 759
unicode-both-case-p 759
unicode-char-equal 760
unicode-char-greaterp 761
unicode-char-lessp 762
unicode-char-not-equal 763
unicode-char-not-greaterp 763
unicode-char-not-lessp 764
unicode-lower-case-p 765
unicode-string-equal 766
unicode-string-greaterp 767
unicode-string-lessp 768
unicode-string-not-equal 769
unicode-string-not-greaterp 770
unicode-string-not-lessp 771
unicode-upper-case-p 772
user-preference 773
when-let 775
when-let* 776
whitespace-char-p 777
with-action-item-error-handling 777
with-action-list-mapping 779
with-unique-names 780

35 The MP Package 783
allowing-block-interrupts 783
barrier-arriver-count 785
barrier-change-count 785
barrier-count 786
barrier-disable 787
barrier-enable 787
barrier-name 788
barrier-pass-through 788
barrier-unblock 789
barrier-wait 790
change-process-priority 793
condition-variable-broadcast 794
condition-variable-signal 794
condition-variable-wait 795
condition-variable-wait-count 796
create-simple-process 797
*current-process* 799
current-process-block-interrupts 800
current-process-in-cleanup-p 801
current-process-pause 801
current-process-unblock-interrupts 803
debug-other-process 804
*default-process-priority* 805
*default-simple-process-priority* 805
ensure-process-cleanup 805
find-process-from-name 806
general-handle-event 807
get-current-process 808
get-process 808
get-process-private-property 809
initialize-multiprocessing 810
*initial-processes* 811
last-callback-on-thread 812
list-all-processes 813
lock-locked-p 814
lock-owned-by-current-process-p 815
lock-recursive-p 815
lock-recursively-locked-p 816
lock-name 816
lock-owner 817
mailbox-empty-p 818
mailbox-peek 819
mailbox-read 820
mailbox-reader-process 821
mailbox-send 821
mailbox-wait-for-event 822
*main-process* 824
make-barrier 824
make-condition-variable 825
make-lock 826
make-mailbox 828
make-named-timer 829
make-semaphore 830
make-timer 831
map-all-processes 832
map-all-processes-backtrace 832
map-process-backtrace 833
map-processes 834
notice-fd 835
process-alive-p 835
process-all-events 836
process-allow-scheduling 836
process-arrest-reasons 837
process-break 838
process-continue 838
process-exclusive-lock 838
process-exclusive-unlock 839
process-idle-time 840
*process-initial-bindings* 841
process-interrupt 842
process-join 843
process-kill 843
process-lock 844
process-mailbox 845
process-name 846
process-p 846
process-plist 846
process-poke 847
process-priority 848
process-private-property 848
process-property 849
process-reset 850
process-run-function 851
process-run-reasons 853
process-run-time 854
process-send 855
process-sharing-lock 856
process-sharing-unlock 857
process-stop 857
process-stopped-p 858
process-unlock 859
process-unstop 860
process-wait 861
process-wait-for-event 862
process-wait-function 862
process-wait-local 863
process-wait-local-with-periodic-checks 865
process-wait-local-with-timeout 867
process-wait-local-with-timeout-and-periodic-checks 868
process-wait-with-timeout 869
process-whostate 870
pushnew-to-process-private-property 871
pushnew-to-process-property 871
ps 872
remove-from-process-private-property 873
remove-from-process-property 873
remove-process-private-property 874
remove-process-property 875
schedule-timer 876
schedule-timer-milliseconds 878
schedule-timer-relative 879
schedule-timer-relative-milliseconds 881
semaphore-acquire 882
semaphore-count 883
semaphore-name 884
semaphore-release 884
semaphore-wait-count 885
simple-process-p 886
syneval-in-process 886
timer-expired-p 887
timer-name 888
unnotice-fd 890
unschedule-timer 890
wait-processing-events 891
with-exclusive-lock 892
with-interrupts-blocked 893
with-lock 894
with-sharing-lock 895
without-interrupts 896
without-preemption 897
yield 897

36 The PARSERGEN Package 899

defparser 899

37 The SERIAL-PORT Package 901

open-serial-port 901
close-serial-port 903
get-serial-port-state 903
serial-port 904
read-serial-port-char 904
read-serial-port-string 905
serial-port-input-available-p 906
set-serial-port-state 907
wait-serial-port-state 907
write-serial-port-char 908
write-serial-port-string 909

38 The SQL Package 911

add-sql-stream 911
attribute-type 912
cache-table-queries 914
map-query 960
*mysql-library-directories* 961
*mysql-library-path* 962
ora-lob-append 963
ora-lob-assign 964
ora-lob-char-set-form 964
ora-lob-char-set-id 965
ora-lob-close 966
ora-lob-copy 967
ora-lob-create-empty 968
ora-lob-create-temporary 969
ora-lob-disable-buffering 970
ora-lob-element-type 971
ora-lob-enable-buffering 971
ora-lob-env-handle 972
ora-lob-erase 973
ora-lob-file-close 974
ora-lob-file-close-all 975
ora-lob-file-exists 975
ora-lob-file-get-name 976
ora-lob-file-is-open 977
ora-lob-file-open 978
ora-lob-file-set-name 978
ora-lob-flush-buffer 979
ora-lob-free 980
ora-lob-free-temporary 981
ora-lob-get-buffer 982
ora-lob-get-chunk-size 984
ora-lob-get-length 985
ora-lob-internal-lob-p 985
ora-lob-is-equal 986
ora-lob-is-open 987
ora-lob-is-temporary 987
ora-lob-load-from-file 988
ora-lob-lob-locator 989
ora-lob-locator-is-init 990
ora-lob-open 991
ora-lob-read-buffer 992
ora-lob-read-into-plain-file 994
ora-lob-read-foreign-buffer 995
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ora-lob-svc-ctx-handle 996</td>
</tr>
<tr>
<td>ora-lob-trim 997</td>
</tr>
<tr>
<td>ora-lob-write-buffer 998</td>
</tr>
<tr>
<td>ora-lob-write-from-plain-file 999</td>
</tr>
<tr>
<td>ora-lob-write-foreign-buffer 1000</td>
</tr>
<tr>
<td>p-oci-env 1002</td>
</tr>
<tr>
<td>p-oci-file 1002</td>
</tr>
<tr>
<td>p-oci-lob-locator 1002</td>
</tr>
<tr>
<td>p-oci-lob-or-file 1002</td>
</tr>
<tr>
<td>p-oci-svc-ctx 1003</td>
</tr>
<tr>
<td>print-query 1003</td>
</tr>
<tr>
<td>query 1004</td>
</tr>
<tr>
<td>reconnect 1005</td>
</tr>
<tr>
<td>restore-sql-reader-syntax-state 1007</td>
</tr>
<tr>
<td>rollback 1007</td>
</tr>
<tr>
<td>select 1008</td>
</tr>
<tr>
<td>simple-do-query 1012</td>
</tr>
<tr>
<td>sql 1014</td>
</tr>
<tr>
<td>sql-connection-error 1015</td>
</tr>
<tr>
<td>sql-database-data-error 1015</td>
</tr>
<tr>
<td>sql-database-error 1016</td>
</tr>
<tr>
<td><em>sql-enlarge-static</em> 1017</td>
</tr>
<tr>
<td>sql-expression 1017</td>
</tr>
<tr>
<td>sql-fatal-error 1018</td>
</tr>
<tr>
<td><em>sql-libraries</em> 1018</td>
</tr>
<tr>
<td><em>sql-loading-verbose</em> 1019</td>
</tr>
<tr>
<td>sql-operation 1019</td>
</tr>
<tr>
<td>sql-operator 1021</td>
</tr>
<tr>
<td>sql-recording-p 1022</td>
</tr>
<tr>
<td>sql-stream 1023</td>
</tr>
<tr>
<td>sql-temporary-error 1024</td>
</tr>
<tr>
<td>sql-timeout-error 1024</td>
</tr>
<tr>
<td>sql-user-error 1024</td>
</tr>
<tr>
<td>standard-db-object 1025</td>
</tr>
<tr>
<td>start-sql-recording 1025</td>
</tr>
<tr>
<td>status 1026</td>
</tr>
<tr>
<td>stop-sql-recording 1027</td>
</tr>
<tr>
<td>table-exists-p 1028</td>
</tr>
<tr>
<td>update-instance-from-records 1028</td>
</tr>
<tr>
<td>update-objects-joins 1029</td>
</tr>
</tbody>
</table>
update-records 1031
update-records-from-instance 1032
update-record-from-slot 1032
update-slot-from-record 1033
with-transaction 1034

39 The STREAM Package 1037

buffered-stream 1037
fundamental-binary-input-stream 1039
fundamental-binary-output-stream 1040
fundamental-binary-stream 1040
fundamental-character-input-stream 1041
fundamental-character-output-stream 1042
fundamental-character-stream 1043
fundamental-input-stream 1043
fundamental-output-stream 1044
fundamental-stream 1044
stream-advance-to-column 1045
stream-check-eof-no-hang 1046
stream-clear-input 1046
stream-clear-output 1047
stream-file-position 1048
stream-fill-buffer 1048
stream-finish-output 1049
stream-flush-buffer 1050
stream-force-output 1051
stream-fresh-line 1051
stream-line-column 1052
stream-listen 1053
stream-output-width 1054
stream-peek-char 1054
stream-read-buffer 1055
stream-read-byte 1056
stream-read-char 1057
stream-read-char-no-hang 1057
stream-read-line 1058
stream-read-sequence 1059
stream-read-timeout 1060
stream-start-line-p 1060
stream-terpri 1061
stream-unread-char 1062
stream-write-buffer 1062
stream-write-byte 1063
stream-write-char 1064
stream-write-sequence 1064
stream-write-string 1065
with-stream-input-buffer 1066
with-stream-output-buffer 1068

40 The SYSTEM Package 1071
apply-with-allocation-in-gen-num 1071
atomic-decf 1072
atomic-incf 1072
atomic-exchange 1073
atomic-fixnum-decf 1074
atomic-fixnum-incf 1074
atomic-pop 1075
atomic-push 1075
augmented-string 1076
augmented-string-p 1076
call-system 1077
call-system-showing-output 1079
cdr-assoc 1081
*check-network-server* 1083
coerce-to-gesture-spec 1083
compare-and-swap 1085
copy-preferences-from-older-version 1086
count-gen-num-allocation 1088
*debug-initialization-errors-in-snap-shot* 1088
default-eol-style 1089
*default-stack-group-list-length* 1090
define-atomic-modify-macro 1090
define-top-loop-command 1092
detect-eol-style 1094
detect-japanese-encoding-in-file 1095
detect-unicode-bom 1096
*directory-link-transparency* 1097
ensure-loads-after-loads 1097
Contents

int32= 1127
int32> 1128
int32>= 1129
int32>> 1129
int32-aref 1130
int32-logand 1131
int32-logandc1 1131
int32-logandc2 1132
int32-logbitp 1133
int32-logeqv 1133
int32-logior 1134
int32-lognand 1135
int32-lognor 1135
int32-lognot 1136
int32-logorcl 1137
int32-logorc2 1137
int32-logtest 1138
int32-logxor 1139
int32-minusp 1139
int32-plusp 1140
int32-to-integer 1140
int32-zerop 1141
integer-to-int32 1142
*line-arguments-list* 1142
load-data-file 1143
locale-file-encoding 1144
low-level-atomic-place-p 1145
make-gesture-spec 1146
make-simple-int32-vector 1151
make-stderr-stream 1152
make-typed-aref-vector 1152
marking-gc 1153
memory-growth-margin 1155
merge-ef-specs 1155
object-address 1156
open-pipe 1157
open-url 1160
pid-exit-status 1161
pointer-from-address 1162
print-pretty-gesture-spec 1163
*print-symbols-using-bars* 1164
product-registry-path 1165
room-values 1166
run-shell-command 1167
safe-locale-file-encoding 1172
set-automatic-gc-callback 1173
set-blocking-gen-num 1174
set-default-segment-size 1177
set-delay-promotion 1178
set-file-dates 1179
set-gen-num-gc-threshold 1180
set-maximum-memory 1181
set-maximum-segment-size 1183
set-memory-check 1184
set-memory-exhausted-callback 1185
set-signal-handler 1187
set-spare-keeping-policy 1189
setup-atomic-funcall 1190
*sg-default-size* 1191
simple-augmented-string 1192
simple-augmented-string-p 1192
simple-int32-vector 1193
*stack-overflow-behaviour* 1193
staticp 1194
storage-exhausted 1194
sweep-gen-num-objects 1195
typed-aref 1196
wait-for-input-streams 1197
wait-for-input-streams-returning-first 1199
with-modification-change 1199
with-modification-check-macro 1200
with-other-threads-disabled 1201

41 Miscellaneous WIN32 symbols 1203
dismiss-splash-screen 1203
*latin-1-code-pages* 1204
long-namestring 1205
*multibyte-code-page-ef* 1205
set-application-themed 1206
Contents

short-namestring 1207
str 1207
lpcstr 1207
lpstr 1207
tstr 1208
lpctstr 1208
lpstr 1208
wstr 1209
lpcwstr 1209
lpwstr 1209

42 The Windows registry API 1211

close-registry-key 1211
collect-registry-subkeys 1212
collect-registry-values 1213
create-registry-key 1215
delete-registry-key 1216
enum-registry-value 1217
open-registry-key 1219
query-registry-key-info 1220
query-registry-value 1221
registry-key-exists-p 1222
registry-value 1223
set-registry-value 1224
with-registry-key 1225

43 The DDE client interface 1227

dde-advise-start 1227
dde-advise-start* 1230
dde-advise-stop 1231
dde-advise-stop* 1232
dde-client-advise-data 1234
dde-connect 1234
dde-disconnect 1235
dde-execute 1236
dde-execute* 1237
dde-execute-command 1237
dde-execute-command* 1238
dde-execute-string 1240
44 The DDE server interface 1257

dde-server-poke 1257
dde-server-request 1258
dde-server-topic 1259
dde-server-topics 1260
dde-system-topic 1261
dde-topic 1261
dde-topic-items 1262
define-dde-dispatch-topic 1263
define-dde-server 1264
define-dde-server-function 1265
start-dde-server 1269

45 Dynamic library C functions 1271

InitLispWorks 1271
LispWorksDlsym 1274
LispWorksState 1275
SimpleInitLispWorks 1276
QuitLispWorks 1277

Index 1279
About this manual
This manual contains a user guide section (previously published separately as the LispWorks User Guide) and a reference section (previously the LispWorks Reference Manual).

User Guide section
The user guide section of this manual describes the main language-level features and tools available in LispWorks, and how to use them.
These chapters describe the central programming tools and features in LispWorks:

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

Preface
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, “Storage Management” covers the behavior (and for wizard level users, configuration) of the garbage collector.
• Chapter 11, “The Profiler” describes a tool for identifying bottlenecks impeding performance of your program.

The next chapter, Chapter 12, “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 13, “LispWorks as a dynamic library” describes how LispWorks operates as a DLL, .dylib or .so.
• Chapter 14, “The Metaobject Protocol” describes how the LispWorks MOP implementation differs from AMOP.
• Chapter 15, “Multiprocessing”, including locks.
• Chapter 16, “Common Defsystem” describes how to use defsystem to combine a series of source files into a manageable project.
• Chapter 17, “The Parser Generator”.
• Chapter 18, “Dynamic Data Exchange” describes how to implement DDE functionality in your Microsoft Windows applications.
• Chapter 19, “Common SQL” explains how to use LispWorks to communicate with databases using SQL.
• Chapter 20, “User Defined Streams” provides an illustrative example showing how to define and implement your own streams.
• Chapter 21, “Socket Stream SSL interface” describes the use of Secure Sockets Layer (SSL) with socket streams.
Chapter 22, “Internationalization” provides an overview of using international characters.

Chapter 23, “LispWorks’ Operating Environment” explains how to find information about the Operating System and how LispWorks was started.

Chapter 24, “64-bit LispWorks” outlines differences between 64-bit LispWorks and 32-bit LispWorks.


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:

- Chapter 25, “The CLOS Package”, describes the LispWorks extensions to CLOS, the Common Lisp Object System.
- Chapter 26, “The COMM Package”, describes the functions providing the TCP/IP interface.
- Chapter 27, “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 28, “The COMPILER Package”, describes symbols available in the COMPILER package.
- Chapter 29, “The DBG Package”, describes symbols available in the DBG package, used to configure the debugging information produced by LispWorks.
Chapter 30, “The DSPEC Package”, describes the symbols available in the DSPEC package, which are used for naming and locating definitions.

Chapter 31, “The EXTERNAL-FORMAT Package”, describes symbols available in the EXTERNAL-FORMAT package.

Chapter 32, “The HCL Package”, describes symbols available in the HCL package.

Chapter 33, “The LINK-LOAD Package”, describes symbols available in the LINK-LOAD package. It applies to LispWorks for UNIX only (not LispWorks for Linux, FreeBSD, or x86/x64 Solaris).

Chapter 34, “The LISPWORKS Package”, describes symbols available in the LISPWORKS package.

Chapter 35, “The MP Package”, describes symbols available in the MP package, giving you access to the multi-processing capabilities of LispWorks.

Chapter 36, “The PARSERGEN Package”, describes symbols available in the PARSERGEN package, the LispWorks parser generator.

Chapter 37, “The SERIAL-PORT Package” documents the Serial Port API. This is implemented only in LispWorks for Windows.

Chapter 38, “The SQL Package” documents symbols used in accessing LispWorks ODBC and SQL functionality.

Chapter 39, “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 40, “The SYSTEM Package”, describes symbols available in the SYSTEM package.

Chapter 41, “Miscellaneous WIN32 symbols”, describes miscellaneous symbols available in the WIN32 package. It applies only to LispWorks for Windows.

Chapter 42, “The Windows registry API”, describes the Windows registry API. It applies only to LispWorks for Windows.
Preface

- Chapter 43, “The DDE client interface”, describes the Dynamic Data Exchange (DDE) client API. It applies only to LispWorks for Windows.
- Chapter 44, “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.

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 27, “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 *LispWorks CAPI User Guide* and the *LispWorks CAPI Reference Manual* describe the CAPI. This is a library of classes, functions, and macros for developing graphical user interfaces for your applications. The *LispWorks CAPI User Guide* is a tutorial guide to the CAPI, and the *LispWorks CAPI Reference Manual* is an in-depth reference text.
Preface


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

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


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

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

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

The LispWorks manuals are also available in HTML format. Commands in the Help menu of any of the LispWorks IDE tools give you direct access to the HTML documentation, using your web browser. Details of how to use these commands can be found in the *LispWorks IDE User Guide*.

Please let us know if you find any mistakes in the LispWorks documentation, or if you have any suggestions for improvements.
Other documentation

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

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

For an account of Metaobject protocols as well as a detailed study of an implementation of CLOS see Kiczales, Rivieres and Bobrow, *The Art of the Meta-Object Protocol*, published by MIT Press, often referred to as AMOP. The LispWorks MOP mostly conforms chapters 5 & 6 of AMOP; the differences are mentioned here in Chapter 14, “The Metaobject Protocol”.

Notation and conventions

Throughout this manual you will find references such as “... the LispWorks file *foo/bar.lisp* ...”. This means a file *bar.lisp* in a subdirectory *foo* of the LispWorks library directory. You can obtain the full path of such a file by evaluating this form in your LispWorks image:

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


Please note that your windows may differ in some respects from the illustrations given in the LispWorks manuals. This is because some details are controlled by the window manager that you are using, not by LispWorks itself.
Preface
Starting LispWorks

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

1.1 The usual way to start LispWorks

On 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 Start menu. On Mac OS X you can run LispWorks by clicking on the ringed "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 Quick Start toolbar in Windows XP.

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 a saved session more convenient than saving an image as described in this section. See “Saved sessions” on page 133 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 call `(load-all-patches)` and then 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 an build script.

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

```
% lispworks-6-0-0-macos-universal -build /tmp/save-config.lisp
```

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

```
C:\Program Files\LispWorks>lispworks-6-0-0-x86-win32.exe -build C:\temp\save-config.lisp
```

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

```
% lispworks-6-0-0-x86-linux -build /tmp/save-config.lisp
```

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

```
% lispworks-6-0-0-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.

1.2.2 Saving a console mode image

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

(...)

The resulting new image, my-console-lispworks, can be made to start the graphical 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 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 302

1.3 Starting the Common LispWorks Graphical IDE

In LispWorks images shipped on the Windows, Mac OS X, Linux, x86/x64 Solaris and FreeBSD 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

To use LispWorks with SLIME you need an image which does not start the LispWorks IDE automatically. Create this image in ~/lw-console as described in “Saving a non-GUI image with multiprocessing enabled” on page 133.

Download SLIME from http://common-lisp.net/project/slime/ and configure Emacs to use "~/lw-console" as the value of inferior-lisp-program as shown in the SLIME README.

3
Starting LispWorks

Note: Use of LispWorks Personal Edition with SLIME is not supported.

1.5 Quitting LispWorks

To quit LispWorks from the graphical 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}

\subsection*{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.

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

\begin{verbatim}
:redo Listener command

:redo &optional command-identifier

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

:get Listener command

:get name command-identifier
\end{verbatim}
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

```lisp
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 738 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 signalled (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 1 > (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 2 > (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.
3 Destroy process.
Type :c followed by a number to proceed
CL-USER 3 : 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` `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`. 
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 :c alone. Similarly, the abort restart can be invoked by entering :a. So an alternative continuation of the division example would be:
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 `hcl:alias` and `hcl:invisible-frame` declarations. See declare, page 398 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 3.5 on page 22).

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 `: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:
3 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.

:\bq  Debugger command

:\bq  \emph{m}

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

### 3.4.2 Moving around the stack

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

:\>  Debugger command

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

:\<  Debugger command

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

:\p  Debugger command

:\p  \emph{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 40 > (compile (defun foo (*zero* one two) (declare
(special *zero*)) (divider one *zero*) (list #'(lambda () one)
two)))
FOO
NIL
NIL

CL-USER 41 > (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, :c <option number> to proceed, or :? for
other options

CL-USER 42 : 1 > :v
Call to FOO (offset 87)
  *ZERO* {Special} : 0
  ONE {Closing} : 1
  TWO : 2

CL-USER 43 : 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.

:l

Debugger Command

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

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

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

\texttt{: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.

\texttt{:cc} \hspace{1cm} \textit{Debugger command}

\texttt{:cc \&optional \texttt{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.

\texttt{: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.

\texttt{:all} \hspace{1cm} \textit{Debugger command}

\texttt{:all \&optional \texttt{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 \texttt{nil}). By default, \texttt{flag} is \texttt{t}.

\texttt{:lambda} \hspace{1cm} \textit{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 \texttt{:dont-know}.
3.4 REPL debugger commands

: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

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

:c

:c &optional m

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

:ret

: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

: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

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

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.

   ```lisp
   (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.

   ```lisp
   (fibonacci "turtle")
   ```

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

3. Type `:bq` at the debugger prompt to perform a quick backtrace. Notice that the problem is in the call to `fibonacci`.

   Note that the calls to `*%apply-interpreted-function` in the backtrace occur because `fibonacci` is being interpreted.

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

4. Do this now, by typing the following form at the debugger prompt.

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

   You intended to call `fibonacci` with the length of the string, but typed in `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. Type `:a` to return to the top level of the debugger.
6. Type `v` to display variable information about the current stack frame in the debugger.

The following output is displayed:

```
M : "turtle"
INDEX : 2
FIB-N-2 : 1
FIB-N-1 : 1
```

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

7. Type in the form below.

```lisp
(setq m (length "turtle"))
```

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

8. Type `:error` to remind yourself of the original error condition you need to handle.

You can handle this error by returning `nil` from the call to `=`, which is the result that would have been obtained if `m` had been correctly set.

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

10. When prompted for a form to be evaluated, type `nil`.

This causes execution to continue as desired, and you can obtain the final result with no further problems.

### 3.5 Debugger control variables

**common-lisp:*debug-io* Variable**

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

**dbg:*debug-print-length* Variable**

The value to which `common-lisp:*print-length*` is bound during output from the debugger.

**dbg:*debug-print-level* Variable**

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

**dbg:*hidden-packages* Variable**

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* Variable**

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 `dbg:set-debugger-options` which may be more convenient.

**dbg:*print-catch-frames* Variable**

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 `dbg:set-debugger-options` which may be more convenient.

**dbg:*print-handler-frames* Variable**

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 `dbg: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 `dbg:set-debugger-options` which may be more convenient.

### dbg:*print-non-symbol-frames*

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

### dbg:set-debugger-options

A call to `set-debugger-options` allows you to set the 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.

The keyword arguments refer to the debugger printing control variables as described below:

- `:all` — affects the state of the `:all` command.
- `:bindings` — `dbg:*print-binding-frames*`  
- `:catchers` — `dbg:*print-catch-frames*`  
- `:hidden` — `dbg:*hidden-packages*`  
- `:non-symbol` — `dbg:*print-non-symbol-frames*`  
- `:handler` — `dbg:*print-handler-frames*`  
- `:restarts` — `dbg:*print-restart-frames*`  
- `:invisible` — `dbg:*print-invisible-frames*`  

Note that the call frames are always displayed, so there is no option to control that.
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 (ini-  
tial value 10).

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

4.2 Inspect

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

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

Table 4.1 Inspector commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>:cv</td>
<td>Display current values of control variables.</td>
</tr>
<tr>
<td>:d</td>
<td>Display current object.</td>
</tr>
<tr>
<td>:dm</td>
<td>Display more of current object.</td>
</tr>
<tr>
<td>:dr</td>
<td>Display rest of current object.</td>
</tr>
<tr>
<td>:h</td>
<td>Display help on inspector commands.</td>
</tr>
<tr>
<td>:im</td>
<td>Recursively invoke a new inspector. <em>m</em> 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 <em>n</em> to value <em>v</em>.</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, <em>int</em>, then the last <em>int</em> 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

`(:dm or :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, :dm does not do anything useful. Typing :d lets you view the object again.

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

### 4.3 Inspection modes

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

```
CL-USER 128 > (inspect "a string with newlines in it")

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

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

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

The asterisk next to

* 1. SIMPLE-STRING

means that SIMPLE-STRING is the current inspection mode.

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

```
CL-USER 130 : Inspect 1 > :m 2

"a
string with
newlines in it" is a SIMPLE-BASE-STRING
0   a
1   string with
2   newlines in it
```

4.3.1 Hash table inspection modes

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

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

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

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

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

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

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

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

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

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

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

CL-USER 1 > (defvar *hash* (make-hash-table))
*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
```
4.3 Inspection modes

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>
```
5

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; however the tracing of some structure accessors and so forth may be lost if the compiler is set to optimize the code for efficiency (so that these calls are inlined).

The standard way of getting functions to be traced in this way is to call the macro `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), by specifying the requisite classes for the arguments in the call to trace. The trace facility handles recursive and nested calls to the functions concerned.

5.1 Simple tracing

This section shows you how to perform simple traces.

1. Type this definition of the factorial function `fac` into the listener:
5 The Trace Facility

(defun fac (n)
  (if (= n 1) 1
    (* n (fac (- n 1)))))

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

(trace fac)

3. Call the function fac as follows:

(fac 3)

The following trace output appears in the listener.

0 FAC > (3)
1 FAC > (2)
  2 FAC > (1)
  2 FAC < (1)
  1 FAC < (2)
0 FAC < (6)

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

- The level of tracing, that is, the number of recursive entries to trace (starting at 0).
- The function name.
- The argument for the current call.

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

Upon exit from each call, the same information is produced: The > symbol denotes entry to a function, and the < symbol denotes exit from it.

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. You can give other expressions to be sent to this stream, in addition to the arguments and results of a function.

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 commands. All tracing can be removed by calling untrace with no arguments.
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 above.

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

*before*  

*before* list of forms

If non-nil, the list of forms is evaluated on entry to the function being traced. The forms are evaluated and the results printed after the arguments to the function.

Here is an example of its use. *traced-arglist* is bound to the list of arguments given to the function being traced. In this example, it is used to accumulate a list of all the arguments to fac across all iterations.

1. In the listener, initialize the variable args-in-reverse as follows:

   ```lisp
   (setq args-in-reverse ())
   ``

2. For the fac function used earlier, set the value of :before to the following list:

   ```lisp
   ((push (car *traced-arglist*) args-in-reverse))
   ```
3. In the listener, evaluate the following form:

\[(\text{fac } 3)\]

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

\textbf{:after}

\textit{Trace keyword}

\texttt{:after list of forms}

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

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

Note also that \texttt{*traced-arglist*} is still bound.

\section*{5.2.2 Evaluating forms without printing results}

\textbf{:eval-before}

\textit{Trace keyword}

\texttt{:eval-before list-of-forms}

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

\textbf{:eval-after}

\textit{Trace keyword}

\texttt{:eval-after list-of-forms}

This option allows you to supply a list of forms for evaluation upon leaving the traced function. The forms are evaluated after printing out
5.2 Tracing options

the results of the function call, but unlike :after their results are not printed.

5.2.3 Using the debugger when tracing

[:break] Trace keyword

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

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

[:break-on-exit] Trace keyword

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

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

[:backtrace] Trace keyword

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

[:quick] Like the :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

:step form

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

5.2.5 Configuring function entry and exit information

:entrycond

:entrycond form

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

:exitcond

:exitcond form

This option controls the printing of information on exit from a traced function. form is evaluated upon exit from the function, and, like :entrycond, information is printed if and only if form evaluates to t. 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.

: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. Type the following call to `fac` in a listener:

   ```lisp
   CL-USER 12 > (fac 10)
   ```

The tracing information printed is as follows:

   ```text
   0 FAC > (10)
   2 FAC > (8)
   4 FAC > (6)
   6 FAC > (4)
   8 FAC > (2)
   9 FAC < (1)
   7 FAC < (6)
   5 FAC < (120)
   3 FAC < (5040)
   1 FAC < (362880)
   ```

5.2.6 Directing trace output

`:trace-output`  

`:trace-output stream`

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

Consider the following example:

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

   ```lisp
   CL-USER 129 > (setq str (open "trace.txt" :direction :output))
   Warning: Setting unbound variable STR
   #<File stream "/u/neald/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:
Inspect the file `trace.txt` in order to see the trace output for the call of `(fac 8)`.

### 5.2.7 Restricting tracing

`:process`  
`Trace keyword`

`:process 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.

`: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 `hcl:*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.
5.2 Tracing options

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

Consider the example below:

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

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

5.2.9 Tracing functions from inside other functions

\texttt{:inside} \hspace{1cm} \textit{Trace keyword}

\texttt{:inside list-of-functions}

The functions given in the argument to \texttt{: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 \texttt{fac2}, which calls \texttt{fac}, as follows:

\begin{verbatim}
(defun fac2 (x)
  (fac x))
\end{verbatim}

2. For the \texttt{fac} function, set the value of \texttt{:inside} to \texttt{fac2}.
3. Call \texttt{fac}, and notice that no tracing information is produced.

\begin{verbatim}
CL-USER 154 > (fac 3)
6
\end{verbatim}

4. Call \texttt{fac2}, and notice the tracing information.
5.3 Example

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

To do this, you specify a `:break` option for $f$ using `trace`. If the form following this option evaluates to a non-nil value upon calling the function, then the debugger is entered. In order to inspect the first argument to the function $f$, you have access to the variable `*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 12 > (defun f (a1 a2) (+ (sqrt a1) a2))
F

CL-USER 13 > (trace (f :break (< (car *traced-arglist*) 0)))
F

CL-USER 14 > (f 9.0 3)
0 F > (9.0 3)
0 F < (6.0)
6.0

CL-USER 15 > (f -16.0 3)
0 F > (-16.0 3)

Break on entry to F
1 (continue) return from break.
2 (abort) return to level 0.
3 return to top loop level 0.
4 Destroy process.

Type :c followed by a number to proceed
```
5.4 Tracing methods

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

1. Type the following methods into the listener:

   (defmethod foo (x)
     (print 'there))

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

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

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

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

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

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

   CL-USER 228 >

5.5 Trace variables

hcl:*max-trace-indent*

Variable

The maximum indentation used during output from trace.
5 The Trace Facility

**hcl:*trace-indent-width* Variable**

The additional amount by which tracing output is indented upon entering a deeper level of nesting.

**hcl:*trace-level* Variable**

The current depth of tracing.

**cl:*trace-output* Variable**

The stream to which tracing sends its output by default.

**hcl:*traced-arglist* Variable**

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

**hcl:*traced-results* Variable**

The variable that holds the results from the traced function.

The following four variables allow the output produced by tracing to be printed in a style that is controlled separately from normal printing:

**hcl:*trace-print-circle* Variable**

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

**hcl:*trace-print-length* Variable**

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

**hcl:*trace-print-level* Variable**

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

hcl:*trace-print-pretty*  

Variable  

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

Note: It can be very dangerous to put advice on system functions.

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:
In the above example you decided to print a message each time **reverse** is called. You called **defadvice** with a description of the function you wanted to alter, a name for the piece of advice, and the keyword **:before** to indicate that you want the code carried out before **reverse** 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 **print-advice** is invoked whenever **reverse** is called, and that it receives the arguments to **reverse**, and that directly after this the original definition of **reverse** is called.

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 **reverse** you could continue the session above as follows:
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 681 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 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 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
6.3 Removing advice

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 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 delete-advice (or the function 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.

```
CL-USER 40 > (delete-advice reverse after-advice)
NIL

CL-USER 41 > (delete-advice reverse print-advice)
NIL
```

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:
6.4 Advice for macros and methods

CL-USER 45 > (progn
  (defclass animal ()
    (genus habitat description
    (food-type :accessor eats)
    (happiness :accessor how-happy)
    (eaten :accessor eaten :initform nil)))
  (defclass cat (animal)
    ((food-type :initform 'fish)))
  (defclass elephant (animal)
    (memory (food-type :initform 'hay)))
  (defmethod feed ((animal animal))
    (let ((food (eats animal)))
      (push food (eaten animal))
      (format t "%-%Feeding -A with -A" animal
              food)))
  (defmethod feed ((animal cat))
    (let ((food (eats animal)))
      (push food (eaten animal))
      (push 'milk (eaten animal))
      (format t "%-%Feeding cat -A with -A and -A" animal food 'milk)))
  (defvar *cat* (make-instance 'cat))
  (defvar *nellie* (make-instance 'elephant)))

NELLIE*

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

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

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

CL-USER 49 > (defadvice
  ((method feed (cat))
   before-feed :before
   (animal)
   (format t "%-%Stroking -A" animal)
   (setf (how-happy animal) 'high))
NIL
CL-USER 50 > (feed *cat*)
Stroking #<CAT 6f35d4>
Feeding cat #<CAT 6f35d4> with FISH and MILK
NIL

CL-USER 51 > (feed *nellie*)
Feeding #<ELEPHANT 71eb7c> with HAY
#<ELEPHANT 71eb7c> has eaten (HAY HAY)

6.5 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 `alpha` that squares a number, and then decide that you intended to return the reciprocal of the square instead. You might proceed as follows.

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

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

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

Note that \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}. 


Finally you decide to alter alpha yet again, this time to produce approximations to $\pi$. $\pi^2/6$ is the sum of the reciprocals of the squares of all the positive integers. So you can generate an approximation to $\pi$ using the sum of the reciprocals of the squares of the integers from one to some limit. (In fact this is not an efficient way of calculating $\pi$, but it could be of interest.)

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

\[
\begin{align*}
\text{(alpha 10.0)} \\
\text{(alpha 100.0)} \\
\text{(alpha 1000.0)} \\
\text{pi}
\end{align*}
\]

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

\[
\begin{align*}
\text{(defun foo (a b c)} \\
\quad \text{(print (list a b c))})
\end{align*}
\]

\[
\begin{align*}
\text{(defadvice (foo and-rest-advice :around) (&rest args)} \\
\quad \text{(format t "advice called with args ~S" args)} \\
\quad \text{(apply #'call-next-advice args))}
\end{align*}
\]

### 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 \texttt{defadvice}.

Pieces of around advice should use the macro \texttt{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 \texttt{delete-advice} or the function \texttt{remove-advice}.
6 The Advice Facility
The dspec system is the machinery underlying the way definitions are named in LispWorks. It supports program development by tracking the locations of definitions, and is also used in tracing and advising functions.

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

This chapter explains the concepts underlying dspecs and their use in tracking locations of definitions. For full details of the programming interface, see Chapter 30, “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:

```
(dsclass-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 qualifiers (specializer*))
```

```
name ::= symbol | (setf symbol)
qualifiers ::= qualifier | (qualifier qualifier*)
specializer ::= symbol | (eql object)
```
Functions in the dspec API accept non-canonical dspecs. All dspec functions, except `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 68 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-undefined` 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 68 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 66 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
(defun saved-value
  (name
   value))
```
The objects are defined using `def-saved-value` and stored on the plist of their name:

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

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

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

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

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

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

```
(dspec: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))))
  :undefiner
  #'(lambda (dspec)
      ;; Remove what def-saved-value recorded
      ~(remprop ,(dspec:dspec-name dspec) 'saved-value))
  :object-dspec
  #'(lambda (obj)
      ;; Given a saved-value object, we can reconstruct its dspec
      (and (saved-value-p obj)
        ~(def-saved-value ,(saved-value-name obj)))))
```

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

```
(dspec:define-form-parser
  (def-saved-value
    {:parser dspec: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 63.

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

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

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

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

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

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

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

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

Define a dspec class for def-computed-saved-value dspecs:
(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:

(dspec: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.

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 def-
var. These cannot be distinguished (other than in the source code), so
\texttt{defparameter} has been defined as a dspec alias for \texttt{defvar}. However, \texttt{defvar}
and \texttt{defconstant} are distinct kinds of variable, since we can easily tell which

type of definition is in effect by calling the function \texttt{constantp}. To define their
dspecs, LispWorks creates a dspec class called \texttt{variable} and uses it as the
superspace argument when defining the \texttt{defvar} and \texttt{defconstant} dspec
classes.

As an explicit example, suppose you have a defining macro

\begin{verbatim}
(defmacro parameterdef (value name)
 `(defparameter ,name ,value))
\end{verbatim}

then

\begin{verbatim}
(ds:define-dspec-alias parameterdef (value name)
 `(defparameter ,name))
\end{verbatim}

would be a suitable appropriate alias definition. This \texttt{define-dspec-alias}
form defines the dspec.

\texttt{define-dspec-alias} is like \texttt{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 \texttt{defparameter} has been found necessary.

\subsection*{7.4.2 Grouping subdefinitions together}

Some definition forms are macros that expand into a group of other
definitions, for example \texttt{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 \texttt{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 \texttt{def}.

\textbf{Note:} to make source location work you will also need a \texttt{define-form-parser}
definition for the macro that expands into the \texttt{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:
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:

- **A pathname** A definition existed in the file named or an editor buffer with that name.

  The keyword **:listener**
  
  A definition was executed interactively in the listener or an editor buffer not associated with a file.

  The keyword **:unknown**
  
  A definition was found in the image (these are entered when a location query does not find any information already in the database).

  The keyword **:implicit**
  
  A definition for a part was recorded, but no information exists for the aggregate.

7.7.2 Recording definitions and redefinition checking

The location information is entered into the database when the definition is executed, by the defining function calling `record-definition`. 
record-definition performs various checks, and returns true or false depending on whether the definition was allowed or not. In particular, it checks if the same name has already been defined in a different location and if so a warning or error can be signalled. See record-definition, page 514 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 63.

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 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 recognise `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`.

```lisp
```
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:

\[
(ds\text{spec}:\text{define-form-parser} \ \text{parameterdef} \ (\text{value} \ \text{name}) \\
\quad \text{`(parameterdef} ,\text{name}))
\]

\[
(ds\text{spec}:\text{define-dspec-alias} \ \text{parameterdef} \ (\text{name}) \\
\quad \text{`(defparameter} ,\text{name}))
\]

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

\[
(\text{parameterdef} \ *\text{foo}*)
\]

### 7.9.2 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:

\[
(\text{defxyz} \ \text{name} \ \text{forms})
\]

gets parsed as

\[
(\text{defxyz} \ \text{name})
\]

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".

### 7.9.3 Reusing form parsers

The form parser established above was specifically for `parameterdef` forms. However if you have other definers of similar syntax - in this example, defin-
ers for which the name is the second subform - then you can define a form parser which can be associated with each of them, as follows:

```
(dspec:define-form-parser (name-second (:anonymous t))
  (value name)
  `((name-second ,name))
```

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

```
(defmacro constantdef (value name)
  `(defconstant ,name ,value))
```

then you can associate the same parser with both this and `parameterdef`:

```
(dspec:define-form-parser (parameterdef
  (:parser name-second-form-parser)))
(dspec:define-form-parser (constantdef
  (:parser name-second-form-parser)))
```

### 7.9.4 Example: defcondition

Suppose you have a macro based on `define-condition`:

```
(defmacro defcondition (&rest args)
  `(define-condition ,@args))
```

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

```
(defcondition foo () ())
```

Two setups are needed to allow the editor to locate such a defining form. Firstly, this tells the system how to parse `(defcondition ...)` toplevel forms:

```
(dspec:define-form-parser
  (defcondition
    (:alias define-condition)))
```

So now:

```
(dspec:parse-form-dspec '(defcondition foo () ()))
=>
(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 ...".

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

So now this definition can be located:

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

just as if it were

\[
\text{(define-condition foo () ()})
\]
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 and undefining action lists

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`.

```lisp
define-action-list
```

`define-action-list uid &key documentation sort-time dummy-actions default-order execution-function`

The `define-action-list` macro defines an action list.

`uid` is a unique identifier, and is a general lisp object, to be compared by `equalp`. It names the list in the global registry of 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 `*handle-existing-action-list*` variable.

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`.

`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...
8.1 Defining and undefining action lists

'(beginning :middle :end)

default-order specifies default ordering constraints for subsequently
defined action-items where no explicit ordering constraints are speci-
fied. An example is

'(:after :beginning :before :end)

exection-function specifies a user-defined function accepting 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 addi-
tional arguments passed to execute-actions, respectively. The remain-
ing arguments are any number of keyword-value pairs that may be
specified in the call to execute-actions. If no execution function is spec-
ified, then the default execution function will be used to execute the
action-list.

undefine-action-list

Macro

define-action-list uid

The macro undefine-action-list flushes the specified list (and all its
action-items). If the action-list specified by uid does not exist, then han-
dling is controlled by the value of the *handle-missing-action-list*
variable.

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 with-action-list-error-handling macro provides a simple mecha-
nism to trap errors and print warnings while executing each action-item.

All execution-functions are required to accept arguments of the form:

(action-list other-args &rest keyword-value-pairs)

where the two required arguments are the action-list and the list of additional
arguments passed to execute-actions (see above), respectively. The remaining
arguments are any number of keyword-value pairs that may be specified in the call to execute-actions. See the LispWorks Reference Manual entries for with-action-list-mapping and with-action-item-error-handling for examples of execution-functions.

Actions are added to an action list using define-action, and are removed using undefine-action.

**define-action**

define-action name-or-list action-name data &rest specs

The macro define-action adds a new action to the list specified by name-or-list, which will be executed according to the action list's execution function.

**undefine-action**

undefine-action name-or-list action-name

The macro undefine-action removes the action specified by action-name from the list specified by name-or-list.

### 8.2 Exception handling variables

The following global variables are used to control the handling of exceptions:

**+handle-existing-action-list+**

The variable +handle-existing-action-list+ is 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 default value is '(:warn :skip). It is used by the define-action-list macro.

**+handle-existing-action-in-action-list+**

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. The default value is ‘(:warn :redefine). It is used by define-action.

*handle-missing-action-list*  
Variable

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. It is used by undefine-action-list, print-actions, execute-actions, define-action and undefine-action.

*handle-missing-action-in-action-list*  
Variable

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 default value is :warn. It is used by undefine-action.

8.3 Other variables

*default-action-list-sort-time*  
Variable

The variable *default-action-list-sort-time* contains 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.

8.4 Diagnostic utilities

Two diagnostic functions are provided: print-actions which prints out the actions on an action list. and print-action-lists, which provides a list of all the defined action lists.
8 Action Lists

print-actions

Function

print-actions name-or-list &optional stream

The function print-actions generates a listing of the action items on this action-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".

print-action-lists

Function

print-action-lists &optional stream

The function print-action-lists generates a listing of all the action lists in the global registry. (The ordering of the action lists here is essentially random.)

8.5 Examples

This example illustrates “typical” use of action lists. The define-action forms might be scattered across several files (mail-utilities.lisp, caffeine.lisp, and so on). Each of the functions, such as read-mail, dont-panic, and so on, take one argument: hassled-p.

(in-package "CL-USER")

(define-action-list "On arrival at office" 
  :documentation "Things to do in the morning"
  :dummy-actions '("Look busy")
  :default-order '(:before "Look busy"))

(define-action "On arrival at office" "Read mail" 'read-mail)

(define-action "On arrival at office" "Greet co-workers"
  'say-hello)

(define-action "On arrival at office" "Drink much coffee"
  'wake-up:after "Locate coffee machine")

(define-action "On arrival at office" "Locate coffee machine"
  'dont-panic)
8.5 Examples

(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-

den.

(defun find-thing (name)
  (do-things #'(lambda (thing)
                (and (equal name (thing-name thing))
                     thing))
             :or))
(defun add-things ()
  (reduce '+ (do-things 'thing-number :collect)))

8.6 Standard Action Lists

The following action lists are defined in LispWorks as shipped:

"When starting image" - Actions to be executed upon image startup.

"Confirm when quitting image" - Actions to be executed before the image
quits. Every action must return non-nil as its first value, otherwise the quit
will be aborted once the actions are complete.

"When quitting image" - Actions to be executed when the image quits, after
success of the "Confirm when quitting image" actions.

"Initialize LispWorks Tools" - Things to do when the LispWorks IDE
starts on a screen. You may customise 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 redis-
playing 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.
The Compiler

The compiler translates Lisp forms and source files into binary code for the host machine. A compiled Lisp function, for instance, is a sequence of machine instructions that directly execute the actions the evaluator would perform in interpreting an application of the original source lambda expression. Where possible the behaviors of compiled and interpreted versions of the same Lisp function are identical. Unfortunately the definition of the Common Lisp language results in certain unavoidable exceptions to this rule. The compiler, for instance, must macroexpand the source before translating it; any side effects of macro-expansion happen only once, at compile time.

By using declarations, you can advise the compiler of the types of variables local to a function or shared across an application. For example, numeric operations on a variable declared as a single-float can be compiled as direct floating-point operations, without the need to check the type at execution time. You can also control the relative emphasis the compiler places on efficiency (speed and space), safety (type checking) and support for debugging. By default the compiler produces code that performs all the necessary type checking and includes code to recover from errors. It is especially important that the type declarations be correct when compiling with a safety level less than 3 (see later in this chapter for more details).

When compiling a Lisp source file, the compiler produces its output in a format that is much faster to load than textual Lisp source — the “fasl” or “fast-
load” form. Fasl files contain arbitrary Common Lisp objects in a pre-digested form. They are loaded without needing to use the expensive \texttt{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 \texttt{eq} before fasl-dumping are created \texttt{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 \texttt{dump-forms-to-file} for details of all the possible fasl file extensions.

\subsection{9.1 Compiling a function}

The function \texttt{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 \texttt{nil}). If the definition is omitted then the current symbol-function of the symbol is used. Below are some examples:

\begin{verbatim}
CL-USER 3 > (compile (defun fred (a b)
                     (dotimes (n a) (funcall b))))
; FRED
FRED
NIL
NIL

CL-USER 4 > (funcall (compile nil '(lambda (n) (* n n))) 7)
; NIL
49

CL-USER 5 > (compile 'ident-fun '(lambda (x) x))
; IDENT-FUN
IDENT-FUN
NIL
NIL
\end{verbatim}
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 *eval-when* special form can be used to force such side effects to take effect at the time of compilation, rather than loading.

9.3 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.
9.4 Compiler control

There are ways to control the nature of compiled code via the `declare` special form and `proclaim` function. See later in this chapter for fuller discussion of these two forms.

In particular there are a set of optimize qualities which take integral values from 0 to 3, in order to control the trade-offs between code size, speed, compilation time, debuggability of the resulting code, and the safety of the code (whether type checks are omitted). 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 signallable error.

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:

```
(proclaim `(optimize (safety 3)))
```

or

```
(proclaim `(optimize safety))
```
and reduce debugging information to a minimum by

(proclaim '(optimize (debug 0)))

Normally code is interruptible, but when going for the extreme levels of speed and “undebuggability” this ceases to be the case unless you also ensure it
thus:

(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 variable when source debugging is on)</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 <code>make-instance</code> and <code>find-class</code> optimizations</td>
</tr>
<tr>
<td>debug=3</td>
<td>Avoids <code>gethash</code> and <code>puthash</code> optimizations</td>
</tr>
<tr>
<td>debug=3</td>
<td>Avoids <code>ldb</code> and <code>dpb</code> optimizations</td>
</tr>
<tr>
<td>debug=3</td>
<td>Avoids an optimization to <code>last</code></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>
</tbody>
</table>
### Combining debug and safety levels in the compiler

<table>
<thead>
<tr>
<th>Keyword settings</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<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>debug&gt;=1</td>
<td>Call count count</td>
</tr>
<tr>
<td>safety&gt;1</td>
<td>Check number of args</td>
</tr>
<tr>
<td>safety&gt;1 or</td>
<td>Check stack overflow</td>
</tr>
<tr>
<td>interruptible&gt;0</td>
<td></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</td>
<td>Fixnum-only arithmetic with errors for</td>
</tr>
<tr>
<td>fixnum-safety=2</td>
<td>non fixnum arguments.</td>
</tr>
<tr>
<td>safety&lt;3 and</td>
<td>No fixnum overflow checks</td>
</tr>
<tr>
<td>fixnum-safety=1</td>
<td></td>
</tr>
<tr>
<td>safety&lt;3 and</td>
<td>No fixnum arithmetic checks at all</td>
</tr>
<tr>
<td>fixnum-safety=0</td>
<td></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>
<tr>
<td>debug=3</td>
<td>Avoids “ad hoc” predicate type transforms</td>
</tr>
<tr>
<td>compilation-speed=3</td>
<td>Reuse virtual registers in very large functions</td>
</tr>
<tr>
<td>debug=3 and safety=3</td>
<td>(declare (type foo x)) and</td>
</tr>
<tr>
<td></td>
<td>(the foo x) ensure a type check</td>
</tr>
<tr>
<td>float=0</td>
<td>Optimize floating point calculations</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
9 The Compiler

off and you supply incorrect types). You can check this by interpreting the code rather than compiling it.

9.4.1 Examples of compiler control

The following function, compiled with safety = 2, does not check the type of its argument because it merely reads:

```lisp
(defun foo (x)
  (declare (optimize (safety 2)))
  (car x))
```

However the following function, also compiled with safety = 2, does check the type of its argument because it writes:

```lisp
(defun set-foo (x y)
  (declare (optimize (safety 2)))
  (setf (car x) y))
```

As another example, interpreted code and code compiled at at low safety does not check type declarations. To make LispWorks check declarations, you need to compile your code after doing:

```lisp
(declaim (optimize (safety 3) (debug 3)))
```

9.5 Declare, proclaim, and declaim

**declare**

Special form

```lisp
declare (declaration *)
```

There are two distinct uses of `declare`, one is to declare Lisp variables as “special” (this affects the semantics of the appropriate bindings of the variables), and the other is to provide advice to help the Common Lisp system (in reality the compiler) run your Lisp code faster, or with more sophisticated debugging options.

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`, etc.
(Consult the syntax definition of each special form to see if it takes declare forms and/or documentation strings.)

For more detail, including some LispWorks extensions to Common Lisp, in the reference entry for declare.

### proclaim

#### Function

proclaim declaration-list

*declaration-list* must be a list of declaration forms to be put into immediate and pervasive effect.

Unlike declare, proclaim is a function that parses the declarations in the list (usually a quoted list, note), 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 that the rest of the file is compiled with these optimization levels in effect. (The other way of doing this is to make appropriate declarations in every function in the file).

proclaim simply returns nil.

### declaim

#### Macro

This is a macro equivalent to proclaim.

Below are some examples:

```
(proclaim `(special *fred*))
(proclaim `(type single-float x y z))
(proclaim `(optimize (safety 0) (speed 3)))
```

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.

Do not forget to quote the argument list if it is a constant list. (proclaim (special x)) attempts to call function special.
9.5.1 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.

9.6 Optimizing your code

Careful use of the compiler optimize qualities described above or special declarations may significantly improve the performance of your code. However it is not recommended that you simply experiment with the effect of adding declarations. It is more productive to work systematically:

1. Use the Profiler, described in Chapter 11, “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.

9.6.1 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 allow you to see information about compiler transformations depending on type information, allocation of floats and bignums, floating point variables, function calls, argument types and so on. Here is a simple example:

```lisp
(defun foo (arg)
  (declare (:explain :variables) (optimize (float 0)))
  (let* ((double-arg (coerce arg 'double-float))
         (next (+ double-arg 1d0))
         (other (* double-arg 1/2)))
    (values next other)))
```

```
;;- Variables with non-floating point types:
;;  ARG OTHER
;;- Variables with floating point types:
;;  DOUBLE-ARG NEXT
```

Note: the LispWorks IDE allows you to distinguish compiler optimization hints from the other output of compilation, and also helps you to locate quickly the source of each hint. For more information see the chapter “The Output Browser” in the LispWorks IDE User Guide.

### 9.6.2 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.

#### 9.6.2.1 Optimized and unoptimized INT32 code

When optimized correctly, the intermediate `int32` objects are not constructed.

In unoptimized code, sequences of operations like

```lisp
(sys:int32+ (sys:int32- a b) (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 \texttt{sys:int32} objects passed as arguments.

### 9.6.2.2 The INT32 API

The INT32 API contains the type \texttt{int32}, a vector type \texttt{simple-int32-vector} and accessor, functions to convert \texttt{int32} to and from integer, some constant \texttt{int32} values, and a full range of operators for mod \(2^{32}\) arithmetic.

You can find all these by evaluating

\begin{verbatim}
(apropos "INT32" "SYSTEM" t)
\end{verbatim}

For details for each, see the entries starting with \texttt{int32} in Chapter 40, “The SYSTEM Package”.

### 9.6.2.3 INT32 Optimization

The optimization works safely but without boxing when possible. You need

\begin{verbatim}
(optimize (float 0))
\end{verbatim}

to get the optimization. This \texttt{float} level affects whether INT32 operations are optimized. This declaration must be placed at the start of a function (not on an inner \texttt{let} or \texttt{locally} form).

In this example the \texttt{safety} level assures a second optimization in \texttt{fli:foreign-typed-aref}:

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

### 9.6.3 Floating point optimization

The \texttt{float} declaration 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)

(defun test (a)
  (declare (optimize (speed 3) (safety 0) (float 0))
    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: calls to +, - and * with more than 4 arguments will not be optimized, even with the declaration described above, so avoid such calls to obtain the best floating point performance

9.6.4 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 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.

9.6.5 Usage of special variables

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 by proclaim or declare.

The speedup will be pretty small overall in most cases, because access to specials is usually a small part of a program. However, if the Profiler identifies some piece of code as a bottleneck, you will want to optimize it, and your optimizations may include proclamation of some variable as global or dynamic.

The three declarations described in this section are extensions to Common Lisp. All declare the symbol to be cl:special, along with other information. These three declarations are mutually exclusive between themselves and cl:special. That is, declaring a symbol with any of these declarations eliminates the other declaration:

- hcl:special-global declares that the symbol is never bound.

In 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.

### 9.6.5.1 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.

### 9.6.6 Stack allocation of objects with dynamic extent

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

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

(declare dynamic-extent) will also optimize these specific calls:
• (make-array n)
• (make-array n :initial-element x) without any other arguments
• (make-foo ...) where make-foo is an inline structure constructor. The default constructor is declared inline automatically when none of the defstruct slot initforms are calls to functions.
• (make-string n :element-type 'base-char)

9.6.7 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 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
9.7 Compiler parameters affecting LispWorks

editor commands, such as Show Paths From. By default, these parameters are all \texttt{t}, which allows you to use all the features of these utilities, at the expense of increasing compilation times.

These variables are initially set to \texttt{t} (in the LispWorks file \texttt{config/a-dot-lispworks.lisp}). To speed up compilation times, you should set these variables to \texttt{nil}. The variables can be controlled as a group by using the function \texttt{toggle-source-debugging}. 
9 The Compiler
10

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 32, “The HCL Package” and Chapter 40, “The SYSTEM Package”.

10.1 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” 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 305.

Garbage collection with a naive algorithm is extremely inefficient. The time required to scan an entire image, which may occupy many megabytes of memory, is prohibitive; especially when the collector must perform the scan in a small, fixed, workspace.

### 10.2 Generations and segments

The LispWorks garbage collector 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 garbage collector 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 garbage collector in more detail. The implementation and the programmatic interface differ between 32-bit and 64-bit LispWorks.

### 10.3 Memory Management in 32-bit LispWorks

This section describes the garbage collector (GC) in 32-bit LispWorks 6.0.

In LispWorks for UNIX and LispWorks for Macintosh, the implementation is not significantly different to that in LispWorks 4.x or LispWorks 5.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.
10.3 Memory Management in 32-bit LispWorks

10.3.1 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 garbage collector function (for example mark-and-sweep on generation 2, 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.

10.3.2 Allocation of objects

Normal allocation is done from a buffer, called the small objects buffer. The Garbage Collector (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.

The minimum and maximum size of free block that the GC uses for the small objects buffer can be set by set-gc-parameters, using the keywords
10 Storage Management

:minimum-buffer-size and :maximum-buffer-size. If the minimum size is too small, the system allocates buffers more frequently, thus slowing the program. Making the minimum too big causes more fragmentation, because small free blocks are not used. There is no easy way to determine the optimal values for the small objects buffer, except by experiment.

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).

10.3.2.1 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 garbage collector itself
- 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).
- Objects allocated by malloc, realloc and memalign in foreign code loaded as above.

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.)
10.3 Memory Management in 32-bit LispWorks

10.3.2.2 Allocation in different generations

Objects that are known to have long life can be allocated directly in a higher generation, by using `allocation-in-gen-num` and `set-default-generation`. Note that both these functions have a global effect, i.e. 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.

See also “Allocation of interned symbols and packages” on page 116 and “Allocation of stacks” on page 117.

10.3.3 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 `mark-and-sweep`.

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.
10.3.4 Garbage collection strategy

When the Garbage Collector runs out of memory, it has to find more memory. Normally (that is, when allocating in generation 0) the first operation is a mark and sweep. Before performing the mark and sweep, the GC compares the amount of memory allocated since the previous mark and sweep with the minimum-for-sweep value, which is set by set-gc-parameters. If the amount allocated is less than minimum-for-sweep the GC does not do a mark and sweep, but causes an overflow (described below). This prevents an excessive number of mark and sweep operations in periods when the program allocates a large amount of data which stays alive.

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 (To change the display would take longer than the mark-and-sweep operation itself.).

Note: the GC monitor window appears only in the Motif IDE.

If more than minimum-for-sweep has been allocated, a mark and sweep operation takes place. After this operation the GC checks that the segment it was trying to allocate to has more free space than the minimum free space for this segment. If the remaining free space is less than minimum-free-space, the GC tries to create more free space by promoting objects from the segment.

Before promoting, the GC performs two checks. First, it checks that there are enough objects marked for promotion to justify a promotion operation. The minimum free space for a segment is set by set-minimum-free-space, and can be shown by (room t).

Second, the GC checks that there is enough free space in the next generation to accommodate the promoted objects. If there is insufficient space, the GC tries to free some, either by a mark and sweep on the next generation, promoting the next generation, or by enlarging the generation.

The minimum amount of space for promotion is the value minimum-for-promote, which is set by set-gc-parameters.

If there is insufficient space, and there are not enough objects marked for promotion, the GC increases the size of the image, by overflow, as described below.
10.3.5 Overflow

If the amount allocated from the previous mark and sweep operation is less than $\text{:minimum-for-sweep}$, 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 $\text{:minimum-overflow}$ and $\text{:maximum-overflow}$, which can be set by $\text{set-gc-parameters}$. 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 $\text{:maximum-overflow}$.

10.3.6 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 if 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 $\text{expand-generation-1}$, which specifies whether expansion or promotion takes place.

If generation 1 is expanded, the amount it tries to expand by is the value $\text{:new-generation-size}$ (set by $\text{set-gc-parameters}$) in words (i.e. multiples of 4 bytes), or the amount of free space needed, whichever is bigger. If $\text{:new-generation-size}$ is 0, it is not expanded. In this case part of the objects marked for promotion are not promoted.

10.3.7 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 $\text{:new-generation-size}$ multiplied by two. Garbage collection of generation 2 can be caused by calling the function $\text{collect-generation-2}$ with appropriate argument.
10.3.8 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.

10.3.9 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, `mark-and-sweep` 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, `try-move-in-generation` is better than `try-compact-in-generation`.

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

```lisp
(defun call-memory-functions()
  (mark-and-sweep 2) ; first collect all dead objects
  (multiple-value-bind (tf tsb tlb)
      (check-fragmentation 2) ; check the fragmentation
    (when (and (> 10000000 tlb)
               (> (ash tf -2) tlb))
      (try-move-in-generation 2 0))))
```
A function such as this can be called at times when a pause of a few seconds is acceptable, and it will keep the memory of generation 2 unfragmented.

It is not possible to give definitive guidance here on how to use `try-move-in-generation` or `try-compact-in-generation`, because it depends on the way the application uses memory. In general, these functions will always improve the behavior of the application. Therefore the main problem is to identify points in the execution of the application where they can be called without causing unacceptably long pauses.

### 10.3.10 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 116. For full details of these functions, see their reference entries.

#### 10.3.10.1 Determining storage usage

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

In 32-bit LispWorks, `memory-growth-margin` returns the amount by which the Lisp heap can grow, if `set-maximum-memory` has been called.

#### 10.3.10.2 Allocating in specific generations

To control the allocation of objects to generations, use `allocation-in-gen-num`, `get-default-generation`, `set-default-generation` and `*symbol-alloc-gen-num*`.

#### 10.3.10.3 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`. 
10.3.10.4 Controlling the garbage collector

The functions that are most likely to be useful for controlling the GC are room, check-fragmentation, gc-generation (replacing mark-and-sweep) 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.

10.4 Memory Management in 64-bit LispWorks

This section describes the garbage collector (GC) in 64-bit LispWorks.

10.4.1 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 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:

```lisp
(room t)
```

Additionally you can see the segments of each generation in the output of:

```lisp
(room :full)
```

After the total allocation in each generation, this prints the allocation type for each segment followed by the hexadecimal address range for allocating objects.

10.4.2 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 \textit{what} argument to the function \texttt{apply-with-allocation-in-gen-num}, are:

\begin{itemize}
  \item [\texttt{:cons}] The segment contains only conses.
  \item [\texttt{:symbol}] The segment contains only symbols (and does not include symbol names or any of the other properties of symbols).
  \item [\texttt{:function}] The segment contains only function objects.
  \item [\texttt{:non-pointer}] The segment contains only objects that do not contain pointers (strings, specialized numeric arrays, double-floats).
  \item [\texttt{:other}] The segment contain other objects, that is any object that contain pointers, and is not a symbol, cons or a function.
\end{itemize}

The derived allocation types are:

\begin{itemize}
  \item [\texttt{:mixed}] The segment contains a mixture of \texttt{:other}, \texttt{:function} and \texttt{:symbol}, but not \texttt{:cons} or \texttt{:non-pointer}.
  \item [\texttt{:cons-static}] The segment contains cons objects that are static.
  \item [\texttt{:non-pointer-static}] The segment contains objects that do not contain pointers and are static (currently stacks are also allocated in these segments).
  \item [\texttt{:mixed-static}] The segment contains a mixture like \texttt{:mixed}, but static.
  \item [\texttt{:weak}] The segment contains weak objects (arrays, and internals of weak hash tables).
  \item [\texttt{:other-big}] The segment contains a single very large simple vector. The vector is static.
  \item [\texttt{:non-pointer-big}]
\end{itemize}
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.

### 10.4.3 Garbage Collection Operations

In 64-bit LispWorks there are two methods of garbage collection: *mark and sweep* (also referred to simply as *mark*) and *copy*. 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 114.

### 10.4.4 Generation Management

In general, higher generations contain objects that live longer and are therefore much less likely to die. Each garbage collection only collects the generations up to some number, and never reclaims the objects in higher generations.

Objects move between generations by being promoted. For most allocation types, this means that the GC copies the objects from a segment in one generation to a segment in a higher generation. For allocation types :other-big and :non-pointer-big, the objects are not actually copied when they are promoted; but instead the whole segment is reattached to the higher generation. The automatic garbage collection promotes objects until they reach the blocking generation.
10.4 Memory Management in 64-bit LispWorks

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.

10.4.5 Tuning the GC

The garbage collector settings are tuned for typical cases, so in general you do not need to change them. If you are considering tuning the GC, contact Lisp Support.

The main tools for seeing how the GC behaves are the macro `extended-time` and periodical calls to `room`.

In the output of `(room t)`, the allocation in each generation is presented according to the allocation type, which may be useful to decide on possible tuning.

`(extended-time forms)` outputs the time spent in garbage collection, whether automatic or called explicitly. The time is shown according to the maximum generation number that was collected and to whether it was a standard garbage collection (automatic and calls to `gc-generation`) or a marking garbage collection (calls to `marking-gc`).

In addition to `room` and `extended-time`, there are also the functions `count-gen-num-allocation`, `gen-num-segments-fragmentation-state`, and `set-automatic-gc-callback`. These 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 11, “The Profiler” for details of that.

10.4.5.1 Interface for tuning the GC

The main interfaces are those which control the blocking generation.

For generations lower than the blocking generation, objects that survive are promoted, and the system does not automatically promote objects to higher
generations. Thus if the application generates long-lived objects, they will accumulate in the blocking generation.

The behavior when the blocking generation grows is controlled by `set-blocking-gen-num` and `set-gen-num-gc-threshold`. It may also be useful to set the maximum segment size with `set-maximum-segment-size`.

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

`gc-generation` can also be used to promote objects to a higher generation than the blocking generation.

It is normally less important to tune the ephemeral segments, that is the segments below the blocking generation. Functions that may be useful include `set-default-segment-size`, `set-spare-keeping-policy` and `set-delay-promotion`.

### 10.5 Common Memory Management Features

This section summarises Memory Management functionality common to all LispWorks 6.0 implementations.

#### 10.5.1 Timing the garbage collector

The macro `extended-time` is useful when timing the garbage collector.

#### 10.5.2 Reducing image size

To reduce the size of the whole image, use `clean-down`.

#### 10.5.3 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 `mark-and-sweep` explicitly to collect them in a timely manner.

### 10.5.4 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.

### 10.5.5 Mapping across all objects

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

### 10.5.6 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.

### 10.5.7 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.

### 10.5.8 Freeing of objects by the GC

Weak arrays and weak hash tables can be used to allow the GC to free objects.
Relevant functions are \texttt{make-hash-table}, \texttt{set-hash-table-weak}, \texttt{set-array-weak}, \texttt{make-array} and \texttt{copy-to-weak-simple-vector}.

For a description of weak vectors see \texttt{set-array-weak}, page 617.

\section*{10.6 Assisting the Garbage Collector}

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

\subsection*{10.6.1 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 \texttt{nil}.

In the code below, if the \texttt{queue-head} cons is promoted to generation \texttt{n}, then all the other conses will also be promoted to generation \texttt{n} eventually, until generation \texttt{n} is collected. This happens even after calls to \texttt{pop-queue} have removed these conses from the queue.
10.6 Assisting the Garbage Collector

(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.

11.1 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.
11 The Profiler

- 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.

11.2 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.

## 11.3 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 124. 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 124.

The function do-profiling is a convenience function which allows you to profile multiple threads using start-profiling and stop-profiling.
11.3.1 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.

11.3.2 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.
11.4 Profiler output

A typical report would be:

profile-stacks called 564 times

Call tree
Symbol seen (%)  
1: MOD 17 (3)  
  2: FLOOR 5 (1)  
  1: EQL 8 (1)  
  1: >= 7 (1)  
1: REALP 2 (0)  
1: + 6 (1)  
1: LENGTH 4 (1)

Cumulative profile summary
Symbol called profile (%) top (%)  
MOD 1000000 17 (3) 8 (1)  
EQL 2000117 8 (1) 8 (1)  
>= 1000001 7 (1) 5 (1)  
+ 1000000 6 (1) 6 (1)  
FLOOR 1000000 5 (1) 5 (1)  
LENGTH 2000086 4 (1) 4 (1)  
REALP 1000001 2 (0) 2 (0)

Top of stack not monitored 93% of the time

The first line means that Lisp was interrupted 564 times by the profiler.

The call tree shows that in 17 of these interrupts (3% of them) the profiler found the function mod on the stack, in 5 of these interrupts it found the function floor on the stack, and so on. Moreover, floor only appears under the mod branch of the tree, which means that each of these times floor was called by mod.

The cumulative profile summary also shows how many times each symbol was found on the stack. Moreover it shows that the function mod was called 1000000 times, the function eql was called 2000117 times, and so on. (Note: this information is not collected on Intel-based platforms by default.) In 17 of these interrupts it found the function mod on the stack, and on 8 of these occasions mod was on the top of the stack. You can deduce that 526 times the function on the top of the stack was none of those reported.
You can control sort order of the cumulative profile summary with `print-profile-list`.

### 11.5 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.

### 11.6 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.

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))
```

### 11.7 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.
11 The Profiler
This chapter gives examples of how to make changes to LispWorks to make it more suitable for use by you and your colleagues.

12.1 Introduction

12.1.1 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 131 describes how to do this.

12.1.2 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 130 describes these initialization files.
12.1.3 Specific customizations

The remainder of this chapter describes some customizations, all of which can be saved in an image or placed in an initialization file, as needed. You can use both techniques: stable code including patches is saved in the image, whilst experimental or fast-loading code is loaded via the initialization file.

12.2 Configuration and initialization files

There are a number of files that contain configuration and initialization information:

12.2.1 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.

12.2.2 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 global preferences” in the *LispWorks IDE User Guide*).
The default location of your home directory varies on Unix systems, but it is typically something like /home. On Windows, the directory is constructed from the environment variables HOMEDRIVE and HOME- PATH. The directory itself has the same name as your user name, so if you log on as john, your home directory might be /home/john on Unix systems or something like C:\Documents and Settings\john on Windows XP.

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.

12.3 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 133 describes how to save a session, which allows restoring your windowing environment as well as your Lisp objects.

12.3.1 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.

12.3.2 The save-image script

Now create a save-image script which is a file save-image.lisp containing something like:
(in-package "CL-USER")
(load-all-patches)
(load #+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 6.0/My LispWorks.app")
    #:cocoa "my-lispworks")

The script shown loads my-configuration.lisp from a temporary directory. You may need to modify this.

12.3.3 Save your new image

The simplest way to save your new image is to use the Application Builder tool in the LispWorks IDE. Start the Application Builder as described in the LispWorks IDE User Guide, enter the path of your save-image script in the Build script pane, and press the Build the application using the script button.

Alternatively you can run LispWorks in a command interpreter and pass your save-image script in the command line as shown below.

- On Macintosh, run in Terminal.app:
  
  mymac$ "~/Applications/LispWorks 6.0/LispWorks.app/Contents/MacOS/lispworks-6-0-0-macos-universal" -build save-image.lisp

  Your new application bundle is saved in /Applications/LispWorks 6.0/My LispWorks.app

- On Microsoft Windows, run in a MS-DOS window:
  
  C:\temp>"C:\Program Files\LispWorks\lispworks-6-0-0-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:

  linux:/tmp$ lispworks-6-0-0-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.

12.3.4 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 initialzation files mentioned in “Initialization files” on page 130.

Note that your newly saved image runs itself, not a saved session.

12.3.5 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-6-0-0-x86-linux -build /tmp/resave.lisp
```

12.4 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 134 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).

12.4.1 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 `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.

12.4.2 What is saved and what is not saved

All Lisp code and data that was loaded into the image or was created in it is saved. This includes all editor buffers, the Listener history and the value of *, ** and ***.

All threads are killed before saving, so any data that is accessible only through a `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.

12.4.3 Saving a session programmatically

You can save a session by calling execute 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.

12.4.3.2 Non-IDE interfaces

If there are non-IDE interfaces on the screen when save-current-session is invoked, those interfaces are destroyed in the first step, and displayed again in the third step. Note that the display will occur in a different thread than the one running the interface before the saving (which was killed in the first step).

If the interface (or any of its children) contains information that is normally destroyed (in some sense) in the destroy-callback, this information can be preserved over a call to save-current-session by defining methods on the generic functions capi:interface-preserving-state-p or capi:interface-preserve-state.

12.4.4 Saving a session using the IDE

You can save a session or set up periodic automatic session saving using the configuration tools in the LispWorks IDE. See "Session saving" in the LispWorks IDE User Guide for details.
12.5 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:

```
(defvar *my-files*
   '("/path/to/foo1"
     "/path/to/fo02"
     "/path/to/fo03")
)
(dolist (file *my-files*)
  (compile-file file :load t))
```

If you also want to open these files in the Editor tool, then you can add this form in your initialization file, after those above:

```
(define-action "Initialize LispWorks Tools"
  "Open My Files"
  '#(lambda (screen)
      (declare (ignore screen))
      (dolist (file *my-files*)
        (ed file))))
```

12.6 Customizing the editor

This section explains some of the customizations you can make to the Editor tool in the LispWorks IDE.

12.6.1 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.
12.6 Customizing the editor

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`.

### 12.6.2 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`.

### 12.6.3 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" #\C-? :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.

### 12.7 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:
   
   ```lisp
   (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 136 for information on controlling how the source code is displayed.

### 12.8 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 signalled 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.

12.9 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 130) to control the initial working directory.

12.10 Using ! for :redo

The default way of redoing the previous command from the command history is via :redo. If you want to use ! (exclamation mark) instead of :redo, add the following to your .lispworks file:

```
(set-macro-character #\!
    #'(lambda (stream char)
        ':redo))
```

You may wish during some sessions to reset ! back to its normal role as a character. To do this, evaluate:

```
(set-syntax-from-char #\! #\@)
```

12.11 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.

12.11.1 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 global preferences” in the LispWorks IDE User Guide for details.

For example, to load the dynamic-completion code every time you start LispWorks, include the following in your initialization file.

```
(require "dynamic-complete")
```
12.11.2 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.

12.12 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*.

12.13 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".

12.13.1 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 `generic.ppd` file 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.
12 Customization of LispWorks
This chapter describes how to create a dynamic library or DLL from LispWorks and discusses use of the library.

13.1 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 then `GetProcAddress`. On other platforms the dynamic library can be loaded by `dlopen` and then `dlsym`.

The dynamic library is usually of file type `dll` on Windows, `dylib` on Macintosh and 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.
13.2 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.

13.2.1 C functions provided by the system

When LispWorks is a dynamic library the functions described in Chapter 45, “Dynamic library C functions” are automatically available. They allow the loading process control over relocation and unloading of the library.

13.2.2 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.

13.2.3 Example

This script saves an image `hello.dll` which is a Windows DLL:
13.3 Initialization of the dynamic library

```
--- 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-6-0-0-x86-win32.exe -build hello.lisp
```

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

(See “Saving a LispWorks image” on page 131 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.

13.3 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`.
13.3.1 Automatic initialization

On 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.

13.3.2 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.

13.4 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 306.
13.5 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.

13.6 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
14

The Metaobject Protocol


All the LispWorks MOP symbols are in the clos package.

There are some discrepancies between LispWorks and AMOP, which are described in this Chapter.

This Chapter also describes some common problems encountered by programmers using the MOP.

14.1 Metaobject features incompatible with AMOP

14.1.1 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.

### 14.1.2 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.

### 14.1.3 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 150 below).

LispWorks does not support user defined methods for the generic function `make-method-lambda`.

### 14.1.4 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.
14.1 Metaobject features incompatible with AMOP

14.1.5 EQL specializers

eql-specializer, eql-specializer-object and intern-eql-specializer are not implemented.
eql specializers in LispWorks are lists.

14.1.6 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.

14.1.7 Method combinations

method-combination objects do not contain the arguments, merely the type. There is a single method-combination object per type.

Therefore the value returned by generic-function-method-combination, and the default value of the :method-combination initarg, and the :method-combination argument processed by ensure-generic-function-using-class are specific only to the type of the method combination.

Also, find-method-combination is not implemented.

14.1.8 Compatible metaclasses

The AMOP defines that the standard primary method for validate-super-class 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`.

### 14.1.9 Inheritance Structure of Metaobject Classes

`funcallable-standard-object` is implemented as defined in AMOP, except that its class precedence list has direct superclasses

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

rather than

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

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

For details, see `funcallable-standard-object`, page 328.

### 14.2 Common problems when using the MOP

#### 14.2.1 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:
14.3 Implementation of virtual slots

(defclass mclass-1 (standard-class) ()
(defclass mclass-2 (standard-class) ()
(defclass a () () (:metaclass mclass-1))
(defmethod validate-superclass ((class mclass-2) (superclass mclass-1)) t)
(defclass b (a) () (:metaclass mclass-2))

Without the validate-superclass method, the last form signals an error because mclass-1 is an invalid superclass of mclass-2.

14.2.2 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.

14.2.3 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.

14.3 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)
      (find-class 'standard-direct-slot-definition)))
(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)  
                (call-next-method))))  
  (call-next-method)))
(virtual-slot-definition-function slotd))
  (return))
  effective-slotd))

;; Underlying access methods for invoking
;; virtual-slot-definition-function.
(defmethod clos:slot-value-using-class
  ((class virtual-metaclass) object slot-name)
  (let ((slotd (find slot-name (class-slots class)
    :key 'slot-definition-name)))
    (if (typep slotd 'virtual-slot-definition)
      (funcall (virtual-slot-definition-function slotd)
        :get
        object)
      (call-next-method))))

(defmethod (setf clos:slot-value-using-class)
  (value (class virtual-metaclass) object slot-name)
  (format t "%- 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))))
14.3 Implementation of virtual slots

(object)
(call-next-method)))

(defmethod 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.

(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))

(defun a-virtual-class-virtual-slot-function
  (key object &optional value)
  (ecase key
    (:get (let ((real-slot (real-slot object)))
      (if (<= 0 real-slot 100)
        (/ real-slot 100.0)
        (slot-unbound (class-of object)
          object
          'virtual-slot))))
    (:set (setf (real-slot object) (* value 100))
      value)
    (:is-set (let ((real-slot (real-slot object)))
      (<= real-slot 100))
      (unset (setf (real-slot object) -1)))))

;; ----------------------- Virtual Slots --------------------

Compile the code above. Then make an object and access the virtual slot:
CL-USER 1 > (setf object (make-instance 'a-virtual-class))
#<A-VIRTUAL-CLASS 2067B064>

CL-USER 2 > (setf (virtual-slot object) 0.75)

setf slot : VIRTUAL-SLOT
0.75

CL-USER 3 > (virtual-slot object)
0.75

CL-USER 4 > (real-slot object)
75.0

Note that when you call (setf real-slot) there is no output since (setf clos:slot-value-using-class) is not called. Compare with (setf virtual-slot).

CL-USER 5 > (setf (real-slot object) 42)
42

Redefine a-virtual-class with :optimize-slot-access nil:

CL-USER 6 > (defclass a-virtual-class ()
  ((real-slot :initarg :real-slot
    :accessor real-slot
    :initform -1)
  (virtual-slot :accessor virtual-slot
    :initarg :virtual-slot
    :allocation :virtual
    :function
    'a-virtual-class-virtual-slot-function))
  (:metaclass virtual-metaclass)
  (:optimize-slot-access nil))
Warning: (DEFCLASS A-VIRTUAL-CLASS) being redefined in LISTENER
  (previously in H:\tmp\vs.lisp).
Warning: (METHOD REAL-SLOT (A-VIRTUAL-CLASS)) being redefined in LISTENER
  (previously in H:\tmp\vs.lisp).
Warning: (METHOD (SETF REAL-SLOT) (T A-VIRTUAL-CLASS)) being redefined in LISTENER
  (previously in H:\tmp\vs.lisp).
Warning: (METHOD VIRTUAL-SLOT (A-VIRTUAL-CLASS)) being redefined in LISTENER
  (previously in H:\tmp\vs.lisp).
Warning: (METHOD (SETF VIRTUAL-SLOT) (T A-VIRTUAL-CLASS)) being redefined in LISTENER
  (previously in H:\tmp\vs.lisp).
#<VIRTUAL-METACLASS A-VIRTUAL-CLASS 21AD908C>
Now the standard accessors call `slot-value-using-class`, so we see output when calling `(setf real-slot)`

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

  setf slot : REAL-SLOT
42
```
LispWorks supports “lightweight” processes. The programming environment, for example, makes extensive use of this mechanism to create separate processes 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.

15.1 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 \texttt{get-current-process} gets the current process, and is the preferred way of doing so. The variable \texttt{*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” on page 182, or it allows itself to be interrupted by calling \texttt{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 \texttt{process-run-function}. This creates a process with the specified name which commences by applying the specified function to arguments. \texttt{process-run-function} returns immediately and the newly created process runs concurrently.

\section*{15.2 The process programming interface}

\subsection*{15.2.1 Creating a process}
To create a new process, use \texttt{process-run-function}.

\subsection*{15.2.2 Finding out about processes}
The system initializes a number of processes on startup. These processes are specified by \texttt{*initial-processes*}. 
The current process is specified by *current-process*. A list of all the current processes is returned by list-all-processes. 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.

**15.2.3 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.

**15.2.3.1 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.

**15.2.3.2 Process priorities in non-SMP LispWorks**

If there is a runnable process with priority P, then no processes with priority less than P will run. When there are runnable processes with equal priority, they will be scheduled in a round-robin manner.

If a process with priority P is running and a blocked process with priority greater than P becomes runnable, the second process will run when the scheduler is next invoked (either explicitly or at the next preemption tick).

To find the priority of a process, use process-priority. This can be changed using change-process-priority.

```
(mp:change-process-priority proc-1 10)
```

Another way to specify the priority is to create the process with process-run-function, passing the keyword :priority:
15 Multiprocessing

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

15.2.4 Interrupting a process

To interrupt a running process, use process-interrupt or process-kill. To break a process and enter the debugger, use process-break.


Note: The process-wait* functions need to be called from the process you want to suspend.

15.2.5 Blocking interrupts

The purpose of blocking interrupts is to prevent a process aborting in the middle of an operation that needs to be completed. A typical example is the cleanup forms of an unwind-protect.

Blocking interrupts does not provide atomicity. Other processes may continue to execute.

Blocking interrupts limits the control that LispWorks has over the processes, so interrupts should not be blocked except when necessary. However, apart from blocking interrupts in a process it does not affect the behavior of the system.

The following macros and functions allow control over blocking interrupts: allowing-block-interrupts, with-interrupts-blocked, current-process-unblock-interrupts and current-process-block-interrupts.

Additionally the macros unwind-protect-blocking-interrupts and unwind-protect-blocking-interrupts-in-cleanups allow your program to prevent interrupts from stopping cleanup forms from completing.
Compatibility note: In LispWorks 5.1 and previous versions, \texttt{mp:without-preemption} and \texttt{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 \texttt{with-other-threads-disabled}.

15.2.6 Old interrupt blocking APIs removed

The macros \texttt{mp:without-interrupts} and \texttt{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 179) or low level atomic operations (see “Low level atomic operations” on page 175).

- Complete operations. This use was designed to ensure that a set of operations completed without being interrupted by \texttt{mp:process-interrupt}, keyboard breaks and so on. See “Blocking interrupts” on page 164 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 \texttt{mp:without-interrupts} but the ideas also apply to uses of \texttt{mp:without-preemption}.

15.2.6.1 Atomic increment

Old:

\begin{verbatim}
(without-interrupts
 (incf *global-counter*))
\end{verbatim}

New: use low level atomic operations.

\begin{verbatim}
(sys:atomic-incf *global-counter*)
\end{verbatim}
15.2.6.2 Atomic push/pop

Old:

\[
\begin{align*}
\text{(without-interrupts)} \\
\quad \text{(push value *global-list*)} \\
\text{(without-interrupts)} \\
\quad \text{(pop *global-list*)}
\end{align*}
\]

New: use low level atomic operations.

\[
\begin{align*}
\text{(sys:atomic-push value *global-list*)} \\
\text{(sys:atomic-pop *global-list*)}
\end{align*}
\]

15.2.6.3 Atomic push/delete

Old:

\[
\begin{align*}
\text{(without-interrupts)} \\
\quad \text{(push value *global-list*)} \\
\text{(without-interrupts)} \\
\quad \text{(setq *global-list* (delete value *global-list*))}
\end{align*}
\]

New: use a lock, because delete cannot be done atomically since it reads more than one object before modifying one of them.

\[
\begin{align*}
\text{(defvar *global-list-lock* (mp:make-lock :name "Global List"))} \\
\text{(mp:with-lock (*global-list-lock*)} \\
\quad \text{(push value *global-list*)}) \\
\text{(mp:with-lock (*global-list-lock*)} \\
\quad \text{(setq *global-list* (delete value *global-list*)})
\end{align*}
\]

15.2.6.4 Atomic plist update

Old:

\[
\begin{align*}
\text{(without-interrupts)} \\
\quad \text{(setf (getf *global-plist* key) value)} \\
\text{(without-interrupts)} \\
\quad \text{(getf *global-plist* key)}
\end{align*}
\]

New: use a lock, because a plist consists of more than one object so cannot be updated with low level atomic operations.
15.2 The process programming interface

(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))

15.2.6.5 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.

(mp:with-lock ((resource-lock resource))
  (when (plusp (resource-free-item-count resource))
    (decf (resource-free-item-count resource))
    (pop (resource-free-items resource))))

15.2.6.6 Atomic access to a cache in a hash table

Old:

(without-interrupts
  (or (gethash value *global-hashtable*)
    (setf (gethash value *global-hashtable*)
      (make-cached-value))))

New: use the hash table lock.

(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.

```
(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))))
```

15.2.7 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.

15.2.7.1 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*`.

15.2.7.2 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 147 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 169.

**Note:** On Windows, Linux, x86/x64 Solaris, FreeBSD and Mac OS X you cannot save a LispWorks image with multiprocessing running.

### 15.2.7.3 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.

```lisp
(load-all-patches)
(load "my-server-code")
(push '(
  "Start Server" () start-my-server)
  mp:*initial-processes*)
(save-image "my-server" :remarks "My Server" :restart-function 'mp:initialize-multiprocessing :environment nil)
```

See `save-image`, page 605 for a description of how to save an image.

### 15.2.8 Values across processes

This section describes ways to pass or read values in another process.
15.2.8.1 Returning a value from another process

Rather than using global variables to pass values between processes, you can use closures instead. For example:

```lisp
(defun send-with-result (process function)
  (let ((remote-result :none))
    (flet ((resultp ()
              (listp remote-result))
          (run-it ()
              (setq remote-result
                    (multiple-value-list (funcall function))))
          (mp:process-send process (list #'run-it))
          (mp:process-wait "Waiting for result" #'resultp)
          (values-list remote-result)))))
```

15.2.8.2 Accessing symbol values across processes

Use `symeval-in-process` to read the value of a dynamically bound symbol in a given process.

`(setf mp:symeval-in-process)` can set the value of such a symbol. `symeval-in-process` is mostly intended for debugging. Do not call it while the thread is actually running.

15.2.9 Stopping and unstopping processes

This section describes a typical way of using `process-stop` and `process-unstop`.

Suppose a pool of "worker" processes is managed by a "manager" process. A process in the worker pool marks itself as available for work, and then calls `process-stop`. The manager process later finds a worker process that is marked as available for work, puts the work in a place known to the worker process, and then calls `process-unstop` on the worker process.

For this scheme to work properly, the check of whether the worker is available needs to include a call to `process-stopped-p`. Otherwise, it is possible for the following sequence of events to occur:

1. A worker marks itself as available.
2. The manager process finds the worker and gives it the work.
3. The manager process calls `process-unstop` on the worker.

4. The worker process proceeds and calls `process-stop`, and never wakes up.

To guard against this possibility, then the manager should call `process-stopped-p` when finding the worker in the second step above. Alternatively, it could check the result of `process-unstop`.

### 15.2.10 Example

The following example allows two (or more) multiplication tables to be printed out simultaneously.

First, the function to print out a multiplication table.

```lisp
(defun print-table (number total stream)
  (do ((i 1 (+ i 1)))
      ((> i total))
    (format stream "~S X ~S = ~S\n" number i (* i number))
    (mp:process-allow-scheduling)))
```

Note the use of `process-allow-scheduling` to allow the process to be interrupted once during each iteration of the do loop.

Now we define the function that calls `print-table` within multiprocessing:

```lisp
(defun process-print-table (name number total)
  (mp:process-run-function name nil
    #'print-table number total *standard-output*))
```

The `nil` argument is used because no keywords are specified.

`process-print-table` can now be called from two separate Listener windows to print out different multiplication tables simultaneously, for example:

```lisp
(process-print-table "t1" 5 50)
```

in one Listener and:

```lisp
(process-print-table "t2" 6 50)
```

in another Listener.
15.3 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.

15.3.1 Immutable objects

Immutable (or read-only) objects such as numbers, characters, functions, pathnames and restarts can be freely shared between threads.

15.3.2 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 174).

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, 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 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**.

Access to packages is atomic.
Note that pathnames cannot be modified, and therefore access to them is always atomic.

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.

### 15.3.3 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 175).

Making several calls to Common Lisp functions that access hash tables will not be atomic overall. 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 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`.

### 15.3.4 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.

However, all of the atomic operations and locking operations in this chapter 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 these operations. The modification check macros `with-modification-change` and `with-modification-check-macro` also take care of this.

### 15.3.5 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 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 "multi-threaded" 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.
15.4 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 15.1:

Table 15.1 Places for which low-level atomic operations are defined

<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><code>(car cons)</code></td>
<td></td>
</tr>
<tr>
<td><code>(cdr cons)</code></td>
<td></td>
</tr>
<tr>
<td><code>(svref sv index)</code></td>
<td>Only <code>simple-vector</code>.</td>
</tr>
<tr>
<td>Structure accessors</td>
<td>The structure must be defined at compile time, and normally there are only macros for atomic operations on structures.</td>
</tr>
<tr>
<td><code>(slot-value object slot-name)</code></td>
<td>See below.</td>
</tr>
</tbody>
</table>

Notes about atomic `slot-value` operations:

1. They ignore the MOP `slot-value-using-class` protocol and can only be used for `:instance` and `:class` allocated slots.

2. They are slower than the atomic operations on other types of object because they have to lock the instance. It may 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, compare-and-swap, define-atomic-modify-macro and setup-atomic-funcall.

You can test whether a place is suitable for use with these operations by the predicate low-level-atomic-place-p.

### 15.5 Aids for implementing modification checks

The macros with-modification-check-macro and with-modification-change provide a way for a body of code to execute and check whether there was any "modification" during this execution, where modification is execution of some other piece of code. This is useful in situations when reading some data out of some data structure is more common than modification, and reading the data involves getting some values that need to be consistent. It makes it possible to ensure consistency of the values without a lock.

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

*modification-place* is a place as defined in Common Lisp (it does not need to be one of the places for atomic locking) which can receive a fixnum. It must be initialized to a fixnum. It must not be modified by any code except with-modification-change.

*with-modification-check-macro* defines a lexical macro (by macrolet) with the name macro-name which takes no arguments, and is used to check if 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.
15.5.1 Example modification check

```lisp
(defun 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 mod-
ifier code.

15.6 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 func-
tion.
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.

### 15.6.1 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:
Programmatically, if the \( \geq \) is true then \( store_2 \) already occurred before \( load_1 \), therefore \( store_1 \) also occurred before \( load_1 \), and \( load_2 \) which happens after \( load_1 \) must happen after \( store_1 \).

On a single CPU that is true. On a computer with multiple CPU cores it can go wrong (that is, \( load_2 \) can happen before \( store_1 \)) because of two possible reasons:

1. \( load_2 \) may happen before \( load_1 \).
2. \( store_2 \) may happen before \( store_1 \).

To guarantee that \( load_2 \) happens after \( store_1 \), both of these possibilities need to be dealt with. Thus the setter has to be:

\[
\begin{align*}
&\text{(setf (my-structure-value-slot *gs*) \( store_1 \) some-value)} \\
&(\text{sys:ensure-stores-after-stores}) \quad \text{; ensure store order} \\
&(\text{setf (my-structure-counter-slot *gs*) \( store_2 \) (incf-counter))}
\end{align*}
\]

and the reader has to be:

\[
\begin{align*}
&(\text{if (> (my-structure-counter-slot *gs*) \( load_1 \) my-counter}) \\
&\text{(progn)} \\
&\quad (\text{sys:ensure-loads-after-loads}) \quad \text{; ensure load order} \\
&\quad (\text{(my-structure-value-slot *gs*) \( load_2 \)} \\
&\quad (.. \text{something else ...}))
\end{align*}
\]

Note that somehow both threads know about \texttt{counter}, and normally will have to synchronize the getting of its value too.

\section*{15.7 Locks}

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

The two main symbols used in locking are the function \texttt{make-lock}, to create a lock, and the macro \texttt{with-lock}, to execute a body of code while holding the specified lock.
A lock has a name (a string) and several other components. The printed representation of a lock shows the name of the lock and whether it is currently locked. Additionally if the lock is locked it shows the name of the process holding the lock, and how many times that process has locked it. For example:

```lisp
#<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 :timeout 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.

### 15.7.1 Features of lock APIs for SMP

Locks can be marked as recursive or not recursive, and they can be made sharing or exclusive. There are APIs for querying whether a lock can be, or actually is, locked recursively.

There are some guarantees about the lock / unlock functions.

#### 15.7.1.1 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`.

### 15.7.1.2 Querying locks


### 15.7.2 Guarantees and limitations when locking and unlocking

In compiled code `process-lock`, `process-exclusive-lock` and `process-sharing-lock` are guaranteed to return if they locked their argument. In other words there will not be any throw between the time they locked the lock and the time they return. That means that in compiled code the next form will at least start executing, and if it is an `unwind-protect` the cleanup forms will at least start executing. (If the code is evaluated, this is not guaranteed.) "Locking" here also means incrementing the count of a lock that is already held by the current thread.

However these functions may throw before locking. For example, in the following code `process-lock` may throw without locking, for example because something interrupts the process by `process-interrupt`:

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

If this call to `process-lock` does throw without locking, then `process-unlock` will be called on a lock that is not locked.

The correct code that guarantees (when compiled) that `process-unlock` is called on exit only when `process-lock` did lock is:

```lisp
(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`:

```lisp
(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`:

```lisp
(mp:process-lock lock)

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

Note: the guarantees described in this section are relevant only in compiled code.

### 15.8 Process Waiting

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.

#### 15.8.1 Specific Process Wait functions

For communication between processes, these are: `mailbox-read`, `process-wait-for-event` 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.

15.8.2 Generic Process Wait functions

The generic Process Wait functions are:

process-wait and process-wait-with-timeout

process-wait-local and process-wait-local-with-timeout


Note: For brevity we sometimes refer to "the *-periodic-checks functions" or "the *-with-timeout functions".

All the generic Process Wait functions take wait-reason and wait-function arguments and potentially also arguments to pass to the wait-function. The *-with-timeout functions mentioned above also take a timeout argument. The *-periodic-checks functions also take a period argument.

The wait-reason is used only to mark the process as waiting for something for debugging purposes. It does not affect the behavior of the functions.

The generic Process Wait functions "wake up" (that is, they simply return to the caller) either when the timeout passed (if they take a timeout argument), or when the wait function returns true. The three pairs of functions mentioned above differ in the mechanism that calls the wait function.

process-wait and process-wait-with-timeout arrange that the "scheduler" will call the wait function when it runs. The "scheduler" is invoked at various points, in an indeterminate process. The advantage of this is that the programmer does not need to worry too much about when the wait function is going to be called. In non-SMP LispWorks (that is, LispWorks 5.1 and earlier) the programmer does not need to worry at all: when some process sets up some-
thing that would make the wait function return true, the waiter process could not run anyway until the setting-up process stopped for some reason (including preemption), by which time the scheduler would have called the wait function if it had not done it before. In SMP LispWorks (that is, LispWorks 6.0 and later), these two processes can run simultaneously, so the delay between the setting up and the scheduling is not necessary. It can be avoided by "poking" the waiting process with `process-poke`, if the waiting process is known, or by invoking the scheduler by `process-allow-scheduling`.

Note: All the specific Process Wait functions record that they wait, and the operations that allow them to continue implicitly "poke" the waiting process.

A large disadvantage of `process-wait` and `process-wait-with-timeout` is that their `wait-function` is called by the "scheduler" in an indeterminate process. That means that the wait function does not see the dynamic environment of the calling process (including error handlers), and cannot be debugged properly. It is also called often, and so it needs to be reasonably fast and not allocate much. In addition, having to call the wait function adds overhead to the system. Therefore in general, if you can achieve the required effect by using either any of the specific wait functions or a `process-wait-local*` function, you should do that and avoid `process-wait` and `process-wait-with-timeout`.

`process-wait-local` and `process-wait-local-with-timeout` do not have all the disadvantages listed above, but their `wait-function` is called only when the process is poked (or at the end of the `timeout`). That means that the programmer does need to worry about when they are called. Typically some other process will set up something, and then poke the waiting process to check if 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.

### 15.8.3 Communication between processes and synchronization

The simplest way to pass a specific event between two processes it to use `process-wait-for-event` on the receiving process, and `process-send` on the sender side. The "event" that is passed is can be any Lisp object. `process-send` and `process-wait-for-event` use a `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 either `mailbox-wait-for-event` and `mailbox-read`. `mailbox-wait-for-event` should be used on processes that may make windows (including any process associated with a CAPI interface), but can be used elsewhere. `mailbox-read` is faster, but if it used on a process with a window it may cause hanging.

`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.

### 15.8.4 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 "knowledgable" 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 186.

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 187.

15.9 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-time-out 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 186.
- Barriers (counting arrivals at a certain point in the code). See “Synchronization barriers” on page 187.
- Counting Semaphores (limiting the number of users of a shared resource). See “Counting semaphores” on page 188.

15.9.1 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 signalled. 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.

### 15.9.2 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:

```lisp
(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`.

### 15.9.3 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.
15.10 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.

For the details, see the reference entries for `make-timer` and `schedule-timer`.

15.10.1 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.

15.10.2 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.

15.11 Process properties

A "process property" is a pair of an indicator and a value that is associated with it for a process.
LispWorks has two kinds of process properties: general and private. These two kinds of properties are stored separately, and the association of indicator/value in each property kind is independent of any in the other property kind.

General properties are stored in the process plist, and can be modified from other processes.

Private properties can only be modified by the current process. Private properties are faster to modify, because the modification does not need to be thread-safe.

Otherwise there is little difference between general and private properties.

`process-plist` and `(setf process-plist)` are not thread safe. In LispWorks 5.1 and earlier the only interface to process properties is `process-plist`, but this does not work well in SMP LispWorks, and so it is deprecated.

There is no parallel to `process-plist` for the private properties.

The general properties are accessed by: `process-property`, `(setf process-property)`, `remove-process-property`, `pushnew-to-process-property` and `remove-from-process-property`.

The private properties are accessed by: `get-process-private-property` (access from other processes), `process-private-property`, `(setf process-private-property)`, `remove-process-private-property`, `pushnew-to-process-private-property` and `remove-from-process-private-property`.

### 15.12 Native threads and foreign code

Support for native threads differs between platforms as described in this section.

#### 15.12.1 Native threads on Windows, Mac OS X, Linux, x86/64 Solaris and FreeBSD

Each Lisp `mp:process` has a separate native thread and in LispWorks 6.0 these threads can run simultaneously.

**Note:** In LispWorks 5.1 and earlier versions, you can have many runnable `mp:process` objects/native threads, but Lisp code can only run in one thread at a time and a lock is used to enforce this. This can limit performance on a
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 `fli:define-foreign-callable` in any thread, without any other setup.

**Note:** In a LispWorks 5.0 executable on Microsoft Windows you must first call `setup-for-alien-threads` before calling back into Lisp, but in LispWorks 5.1 and later versions this setup is handled automatically (and `setup-for-alien-threads` does not exist).

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.

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

### 15.12.3 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`.

### 15.13 Example

The following is an informal example of multi-processing with a single process (other than the idle process), namely a top-loop. Once it has started up, try `(mp:ps)`.
(in-package "CL-USER")

;;; (guarantee-processes) will start up
;;; multiprocessing with a top-level loop
;;; in this example,
;;; use *base-process* to ensure that base
;;; process will only be pushed
;;; onto *initial-processes* once, no matter how
;;; many times guarantee-processes is called

(defvar *base-process*
  '(*base-process* nil base-process-function))

;;; the base process consists of a top-level
;;; loops with restarts which allow control of
;;; return in the event of an error -- to see
;;; these in action, evaluate (guarantee-processes)
;;; and then an unbound variable.

;;; Note that starting and stopping multiprocessing is not
;;; relevant if the LispWorks IDE is already running. This example
;;; is included for illustration only.

(defun base-process-function ()
  (with-simple-restart
    (abort "Return from multiprocessing")
    (loop
     (with-simple-restart
      (abort "Return to top-level-loop")
      (system:%top-level)))
    (mp::stop-multiprocessing)))

;;;; simple startup of multiprocessing with one
;;;; process (apart from the idle process)

(defun guarantee-processes ()
  (unless mp:*multiprocessing*
    (pushnew *base-process*
      (mp:*initial-processes*)
    (mp:initialize-multiprocessing)))
16

Common Defsystem

16.1 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 Common LispWorks system tools, and the system browser in particular, 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 \texttt{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.

\section*{16.2 Defining a system}

A system is defined with a \texttt{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:

\begin{verbatim}
   CL-USER 5 > (compile-system 'debug-app :force t)
\end{verbatim}

would compile every file in a system called \texttt{debug-app}.

\textbf{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 \texttt{defsys.lisp} which usually resides in the same directory as the members of the system.

The full syntax of \texttt{defsystem} is given in \texttt{defsystem}, page 689. Below is a brief introduction.
16.2 Defining a system

16.2.1 DEFSYSTEM syntax

defsystem

defsystem system-name options &key members rules

- system-name: A symbol used as the name of the system. If a string is given, it is interned in the current package.
- options: Any of a number of options that can be specified.
- members: The members of the system. These may be files of Common Lisp source code, foreign source code, or other systems.
- rules: A set of rules describing the requirements for and order in which compilation and loading of the system members should take place.

See the following sections for more information about these parameters.

16.2.2 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.

16.2.3 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/neald/foo.lisp is 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
16.2.4 DEF SYSTEM 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.
  
  This is an action-member description like :compile "foo". The member can be an actual member of the system or :all (meaning the rule should apply to each member of the system).

- The actions that the target(s) are :caused-by.
  
  The actions that cause the rule to execute successfully.
This is a list of action-member descriptions. The member of each of these descriptions should be either a real system member, or :previous, which means all members listed before the member of the target in the system description.

If any of these descriptions are already in the current plan (as a result of other rules executing successfully, or as a result of default system behavior), they trigger successful execution of this rule.

The actions that the target(s) :requires.

The actions that need to be performed before the rule can execute successfully.

This is a list of action-member descriptions that should be planned for before the action on the target(s). Again, each member should either be a real member of the system, or :previous.

The use of the keyword :previous means, for example, that you can specify that in order to compile a file in the system, all the members that come before it must be loaded.

When the action and member of a target are matched during the traversal of the list of members, the target is inserted into the plan if either of the following are true:

- any of the action-member descriptions in the :caused-by clause is already in the plan, or
- any implicit conditions (such as the source file being newer than the object file) are satisfied.

If the target is put into the plan then other targets are inserted beforehand if the action-member description of any :requires clause is not already in the plan.

### 16.2.5 Examples

Consider an example system, demo, defined as follows:
(defsystem demo (:package "USER")
  :members ("parent"
            "child1"
            "child2")
  :rules ((:in-order-to :compile (*child1* *child2*)
        (:caused-by (:compile "parent"))
        (:requires (:load "parent"))))

This system compiles and loads members into the USER package if the members themselves do not specify packages. The system contains three members — parent, child1, and child2 — which may themselves be either files or other systems. There is only one explicit rule in the example. If parent needs to be compiled (for instance, if it has been changed), then this causes child1 and child2 to be compiled as well, irrespective of whether they have themselves changed. In order for them to be compiled, parent must first be loaded.

Implicitly, it is always the case that if any member changes, it needs to be compiled when you compile the system. The explicit rule above means that if the changed member happens to be parent, then every member gets compiled. If the changed member is not parent, then parent must at least be loaded before compiling takes place.

The next example shows a system consisting of three files:

(defsystem my-system
  (:default-pathname "~/junk/")
  :members (*a* *b* *c*)
  :rules ((:in-order-to :compile (*c*)
        (:requires (:load "a"))
        (:caused-by (:compile "b"))))

What plan is produced when all three files have already been compiled, but the file 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:

```lisp
(defun my-system ()
  (in-order-to :compile :all
    (:requires (:load :previous))))

(let ((defsystem my-system (:load t)))
  (compile-system my-system :load t))
```
Common Defsystem
17 The Parser Generator

17.1 Introduction
The parser generator generates an LALR parser from a specification of a grammar. The parser generator has a simple facility for the static resolution of ambiguity in the grammar and supports an automatic run-time error correction mechanism as well as user-defined error correction. Semantic actions can be included in the rules for the grammar by specifying Lisp forms to be evaluated when reductions are performed.

For further details on LALR parsing, see Compilers, Principles Techniques and Tools, by Aho, Sethi and Ullman, publishers Addison Wesley, 1986.

Load the parser generator by (require "parsergen").

17.2 Grammar rules
The parser generator is accessed by the macro defparser, described below:

```
defparser name {rules}*
```

Macro
name

The name to be used for the parsing function. The remainder of the macro form specifies the reduction rules and semantic actions for the grammar.

rules

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:

```
((non-terminal (grammar-symbol)*) {form}*)
```

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.

### 17.2.1 Example

If a grammar contains the production:

```
tuple -> expression operator expression
```

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

```
((expression expression operator expression) (list $1 $2 $3))
```

Error productions of the form:

```
((nt :error) (some error behavior))
```

are explained in the section below.

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

```
((nt nt1) $1)
```

where the non-terminal `nt` has no other productions and `nt1` serves as the main “top-level” non-terminal.
17.2.2 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} -> \textit{if} \textit{expression} \textit{then} \textit{statement} \textit{else} \textit{statement}
- Rule b: \textit{statement} -> \textit{if} \textit{expression} \textit{then} \textit{statement}

this results in a conflict in the parser between a shift of \textit{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} \ldots \textit{then} \ldots \textit{else} \ldots \textit{statement} being reduced, and a syntax error is produced for the \textit{else} part.

During parser generation, any conflicts between rules are reported, together with information about how the conflict was resolved.

17.3 Functions defined by defparser

The form \texttt{(defparser name grammar)} defines a number of functions. The main function \texttt{name} is defined as the parsing function. For example:

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

defines the function

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

\texttt{lexer} specifies the lexical analyzer function to be used. The optional argument \texttt{symbol-to-string} should be a function mapping grammar symbols to strings for printing purposes. The default value of \texttt{symbol-to-string} is the function \texttt{cl:identity}.
defparser also defines functions corresponding to the individual actions of
the parser.

Normal actions are named:

    name-action\text{index}

and error actions are named:

    name-error-action\text{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.

17.4 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 \rightarrow :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.

17.5 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 (e.g. pushing extra tokens) is done within the parser genera-
tor and the lexical analyzer need not concern itself with this.

17.6 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.

(defparser english-parser
  ((bs s) $1)
  ((s np vp)
   `($1 ,$2))
  ((np :noun) $1)
  ((vp :verb np locp)
   `($1 ,$2 ,$3))
  ((vp :verb locp)
   `($1 ,$2))
  ((vp :verb np)
   `($1 ,$2))
  ((vp :verb)
   $1)
  ((np :art bnp locp)
   `($1 ,$2 ,$3))
  ((np :art bnp)
   `($1 ,$2))
  ((np bnp) $1)
  ((locp :loc np)
   `($1 ,$2)))

;;; The lexer function.

;;; The basic lexing function

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

;;; Getting syntactic categories.

(defparameter *words*
  '((the :art) (a :art) (some :art) (ate :verb) (hit :verb)
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)))))))))
```
The Parser Generator
18 Dynamic Data Exchange

18.1 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.

18.1.1 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.
18.1.2 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.

18.1.3 Advise loops

An advise loop 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.

18.1.4 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:

\[ \text{[command}(\text{arg1, arg2, ...})] \]

18.2 Client interface

18.2.1 Opening and closing conversations

A LispWorks client can open a conversation by using \texttt{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 \texttt{dde-disconnect}.

Another method for managing conversations uses \texttt{with-dde-conversation} to bind a conversation with a server across a body of code. If no conversation is available for \texttt{with-dde-conversation}, then one is automatically opened. The code is executed and the conversation is closed after the body of code exits.

18.2.2 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*.

18.2.3 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.

18.2.3.1 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.

(defclass my-conv (dde-client-conversation) ()

The function 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-advertise-start* :foo "file1" "slot1")
```

When the value of the item specified by "slot1" changes, the server calls
`dde-client-advertise-data` which returns a string, as described above.

The function argument of `dde-advertise-start` and `dde-advertise-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-advertise-data` by default. A different
function can be provided, and should have a lambda list similar to the follow-
ing:

```
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-advertise-stop` or `dde-advertise-stop*`.

### 18.2.4 Request and poke transactions

LispWorks clients can issue request and poke transactions across a conversa-
tion using `dde-request` and `dde-poke`, which take a `conversation` (or a `service
designator/topic` designator pair in the case of an automatically managed con-
versation), and an `item` as their main arguments. In the case of a poke transac-
tion, 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 conversa-
tions) can perform request and poke transactions. It performs a request trans-
action when read, and a poke transaction when set.
18.2.5 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 211). 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:

```
(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:

```
(dde-execute-command* :excel :system 'open '("foo.xls"))
```

18.3 Server interface

18.3.1 Starting a DDE server

To provide a LispWorks application with a DDE server, follow the following three steps.

1. Define a specialized Lisp DDE server class using define-dde-server. Here the server class is called foo-server and it has the service name "FOO":

   ```lisp
   (define-dde-server foo-server "FOO")
   ```
2. Provide the server class with the functionality it requires by specializing methods on it and/or using define-dde-server-function. Here the server function is bar, which takes a string as an argument, and prints this to the standard output. For convenience, the system topic is used, though usually it is better to define your own topic.

   (define-dde-server-function (bar :topic :system)
     :execute
     ((x string))
     (format t "~&~s~&" x)
     t)


   (start-dde-server 'foo-server)

   This function returns the server object, which responds to requests for conversations with the service name "FOO", and accepts execute transactions for the function bar in the "System" topic.

18.3.2 Handling poke and request transactions

Poke and request transactions issued to a server object are handled by defining a method on each of the generic functions dde-server-poke and dde-server-request.

18.3.3 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 inter-
   face. However, a mechanism is provided to extend the functionality of
   the system topic by handling additional items.

18.3.3.1 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 docu-
   ment, 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.

18.3.3.2 Dispatching topics
   A dispatching topic is a topic which has a fixed name and always exists. Dis-
   patching 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.

18.3.3.3 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:
18.3 Server interface

**SZDDESYS_ITEM_TOPICS**

`Constant`

The constant \texttt{SZDDESYS\_ITEM\_TOPICS} has the value "Topics". Referring to this item in the system topic calls \texttt{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 \texttt{:unknown}.

**SZDDESYS_ITEM_SYSITEMS**

`Constant`

The constant \texttt{SZDDESYS\_ITEM\_SYSITEMS} has the value "SysItems". Referring to this item in the system topic calls \texttt{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 \texttt{dde-system-topic} returning the complete list of supported topics. The server can return \texttt{:unknown} if this item is not to be implemented.

**SZDDESYS_ITEM_FORMATS**

`Constant`

The constant \texttt{SZDDESYS\_ITEM\_FORMATS} has the value "Formats", and returns \texttt{unicodetext} and \texttt{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 Chapter 38, “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.

19.1 Introduction

This chapter covers the following areas:

- Initialization and Connection
- The Functional SQL Interface
- The Object-Oriented (CLOS) SQL Interface
- The Symbolic SQL Syntax
- SQL I/O Recording
- SQL Interface Errors

The LispWorks SQL interface uses the following database terminology:
Data Definition Language (DDL)
The language used to specify and interrogate the structure of the database schema.

Data Manipulation Language (DML)
The language used for retrieving and modifying data. Also known as query language.

- **table** A set of records. Also known as relation.
- **attribute** A field of information in the table. Also known as column.
- **record** A complete set of attribute values in the table. Also known as tuple, or row.
- **view** A display of a table configured to your own needs. Also known as virtual table.

### 19.1.1 Overview

Common SQL is designed to provide both embedded and transparent access to relational databases from the LispWorks environment. That is, SQL/relational data can be directly manipulated from within Lisp, and also used as necessary when instantiating or accessing particular Lisp objects.

The SQL interface allows the following:
- Direct use of standard SQL statements as strings
- Mixed symbolic SQL and Common Lisp expressions
- Implicit SQL invocation when instantiating or accessing CLOS objects

The SQL interface provides these features through two complementary layers:
- A **functional** SQL interface
- An **object-oriented** SQL interface

The functional interface provides users with Lisp functions which map onto standard SQL DML and DDL commands. Special iteration constructs which utilize these functions are also provided. The object-oriented interface allows users to manipulate database views as CLOS classes via `def-view-class`. 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.

19.1.2 Supported databases

Common SQL supports connections to various databases on the platforms indicated below in Table 19.1. Common SQL may work with other platform/ODBC driver combinations and we would be pleased to hear of your experience with these. The keyword shown in the last column is the corresponding value of the database-type argument to connect:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Database</th>
<th>Driver/Client library</th>
<th>database-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>ODBC</td>
<td>Microsoft SQL Server</td>
<td>:odbc</td>
</tr>
<tr>
<td>Windows</td>
<td>ODBC</td>
<td>Oracle</td>
<td>:odbc</td>
</tr>
<tr>
<td>Windows</td>
<td>ODBC</td>
<td>Postgres</td>
<td>:odbc</td>
</tr>
<tr>
<td>Linux</td>
<td>ODBC</td>
<td>MySQL</td>
<td>:odbc</td>
</tr>
<tr>
<td>Linux</td>
<td>ODBC</td>
<td>Postgres</td>
<td>:odbc</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>ODBC</td>
<td>MySQL</td>
<td>:odbc</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>ODBC</td>
<td>Postgres</td>
<td>:odbc</td>
</tr>
<tr>
<td>Linux</td>
<td>Oracle</td>
<td>Oracle 9i (r2) or 10g</td>
<td>:oracle</td>
</tr>
<tr>
<td>Mac OS X/Intel/32-bit</td>
<td>Oracle</td>
<td>Oracle 10g</td>
<td>:oracle</td>
</tr>
<tr>
<td>Mac OS X/Intel/64-bit</td>
<td>Oracle</td>
<td>Oracle 10g</td>
<td>:oracle</td>
</tr>
<tr>
<td>Mac OS X/PPC/32-bit</td>
<td>Oracle</td>
<td>Oracle 10g</td>
<td>:oracle</td>
</tr>
<tr>
<td>Windows</td>
<td>Oracle</td>
<td>Oracle 9i (r2) or 10g</td>
<td>:oracle</td>
</tr>
<tr>
<td>Solaris/Intel</td>
<td>Oracle</td>
<td>Oracle 10g</td>
<td>:oracle</td>
</tr>
</tbody>
</table>
Table 19.1  Supported databases

<table>
<thead>
<tr>
<th>Platform</th>
<th>Database</th>
<th>Driver/Client library</th>
<th>database-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solaris/SPARC</td>
<td>Oracle</td>
<td>Oracle 9i (r2) or 10g</td>
<td>:oracle</td>
</tr>
<tr>
<td>HP-UX</td>
<td>Oracle</td>
<td>Oracle 9i (r2) or 10g</td>
<td>:oracle</td>
</tr>
<tr>
<td>Linux</td>
<td>Oracle</td>
<td>Oracle</td>
<td>:oracle8</td>
</tr>
<tr>
<td>Solaris/SPARC</td>
<td>Oracle</td>
<td>Oracle</td>
<td>:oracle8</td>
</tr>
<tr>
<td>HP-UX</td>
<td>Oracle</td>
<td>Oracle</td>
<td>:oracle8</td>
</tr>
<tr>
<td>Linux</td>
<td>PostgreSQL</td>
<td>Postgres</td>
<td>:postgresql</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>PostgreSQL</td>
<td>Postgres</td>
<td>:postgresql</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>PostgreSQL</td>
<td>Postgres</td>
<td>:postgresql</td>
</tr>
<tr>
<td>Windows</td>
<td>PostgreSQL</td>
<td>Postgres</td>
<td>:postgresql</td>
</tr>
<tr>
<td>Solaris/Intel</td>
<td>PostgreSQL</td>
<td>Postgres 8.2</td>
<td>:postgresql</td>
</tr>
<tr>
<td>Linux</td>
<td>MySQL</td>
<td>MySQL</td>
<td>:mysql</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>MySQL</td>
<td>MySQL</td>
<td>:mysql</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>MySQL</td>
<td>MySQL</td>
<td>:mysql</td>
</tr>
<tr>
<td>Windows</td>
<td>MySQL</td>
<td>MySQL</td>
<td>:mysql</td>
</tr>
<tr>
<td>Solaris/Intel</td>
<td>MySQL</td>
<td>MySQL</td>
<td>:mysql</td>
</tr>
<tr>
<td>Solaris/SPARC</td>
<td>MySQL</td>
<td>MySQL</td>
<td>:mysql</td>
</tr>
</tbody>
</table>

**Note:** MySQL versions prior to 4.1.1 should be run in ANSI mode to work with Common SQL. That is, mysql 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.
19.2 Initialization

The initialization of Common SQL involves three stages. Firstly the SQL interface 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.

19.2.1 SQL interface

The SQL interface itself is loaded by issuing the command (require "odbc") or (require "oracle") or (require "postgresql") or (require "mysql"). In an application, this step should be performed at build-time.

Not all of these modules are available on every LispWorks platform. See Table 19.1, page 221 for information about which databases are supported per-platform.

19.2.2 Database classes

A connection to a database is represented by a CLOS instance which holds information about the connected database. The special variable *default-database* holds the current connection. The database class is subclassed on both vendor and version to provide the right kind of specialized behavior across database facilities: for example, the transaction model or the “brand” of SQL.

19.2.3 Initialization functions and variables

The initialization of the chosen database type is achieved by calling initialize-database-type with the appropriate value of database-type. In an application, this step should be done at runtime. Where multiple database types are supported, it is possible to initialize more than one database type if needed (by making multiple calls to initialize-database-type).
The following functions and variables are relevant to initialization:

*default-database-type* specifies the default type of database. The possible values are shown in the database-type column of Table 19.1, page 221.

The function initialize-database-type initializes a database type according to the value of its database-type argument, which defaults to the value of *default-database-type*.

A sample code sequence for initializing Common SQL to work with an ODBC database, using the above functions and variables, is as follows:

```
(requires *odbc*)
(setf *default-database-type* :odbc)
(initialize-database-type)
```

You can find which database types have been initialized by the value of the variable *initialized-database-types*.

### 19.2.4 Database libraries

**Note:** This section applies only to Unix/Linux systems.

A database directory environment variable specifies the root of the database directories. This variable is checked by LispWorks when you initialize a database type, and the libraries loaded are dependent on its value. The details of foreign code loading are described in the *LispWorks Foreign Language Interface User Guide and Reference Manual*.

Note that most users only need to set the appropriate environment variable for their specific database vendor.

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*.

In LispWorks for UNIX only (not LispWorks for Linux, x86/x64 Solaris, FreeBSD or Macintosh), the list of library modules is added to *link-load:*default-libraries* and *read-foreign-modules* is called to do the loading. If you need to load a different set of library modules, add your list of library modules to *link-load:*default-libraries* before loading.
19.2.5 General database connection and disconnection

Once the database type has been initialized a connection can be established by calling `connect` with an appropriate `connection-spec`. A call to `connect` sets `*default-database*` to the database instance which represents the connection. All the other database functions described take a `:database` argument that can be either a database or a database name, and which defaults to the value of `*default-database*`.

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`.

19.2.5.1 Connection example

The following example assumes that the `:odbc` database type has been initialized as described in “Initialization functions and variables” on page 223. It connects to two databases, `scott` and `personnel`, and then prints out the connected databases.

```lisp
(setq *default-database-type* :odbc)
(connect "scott")
(connect "personnel" :database-type :odbc)
(print *connected-databases*)
```

19.2.6 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.

### 19.2.7 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.

#### 19.2.7.1 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").

#### 19.2.7.2 Connecting to ODBC using a plist

In the plist, the acceptable keywords are :`username`, :`password`, :`dsn` and :`connection`.

:`connection` is a synonym of :`dsn`. 
19.2 Initialization

19.2.8 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:

- **username**: anonymous user
- **password**: No password
- **dbname**: No default database
- **hostname**: localhost
- **port**: 3306 (unless using unix-socket).

19.2.8.1 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:

```
me/my-password/my-db@:3307
```

`hostname` may also specify a unix socket name, which must start with the character `/`.

`dbname` may be omitted along with the preceding `@`.

`password` may be omitted. If `dbname` is also omitted, the preceding `@` can be omitted too.

`username` may be omitted.
19.2.8.2 Connecting to MySQL using a plist

A connection-spec can be a plist containing (some of) the keywords :username, :password, :dbname, :hostname, :port, :connection, :unix-socket. Each of these keywords may be omitted.

If :unix-socket is specified, then none of :hostname, :port and :connection can be specified. If :hostname is specified then :connection must not be specified. The value supplied for :hostname can be a raw hostname, or a string of the form hostname:port. If :connection is specified then it can 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.

19.2.8.3 Locating the MySQL client library

The MySQL interface to initialize, it must find the appropriate MySQL client library. The special variables *mysql-library-path* and *mysql-library-directories* give you control over this.

19.2.8.4 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:

```bash
gcc -dynamiclib -fno-common -o /usr/local/mysql/lib/libmysqlclient_r.dylib -all_load /usr/local/mysql/lib/libmysqlclient_r.a -lz
```
19.2 Initialization

To build the 64-bit dynamic library:

```bash
gcc -m64 -dynamiclib -fno-common \
-o /usr/local/mysql/lib/libmysqlclient_r.dylib \
-all_load /usr/local/mysql/lib/libmysqlclient_r.a -lz
```

This command should be executed as the root user, or some other user with write permission to the `/usr/local/mysql/lib` directory and assumes that MySQL was installed in `/usr/local/mysql`, which is the location used by the prepackaged downloads.

An alternate way to create a dynamic library is to build MySQL from its source code with the `--enable-shared` flag.

By default, LispWorks expects to find the library either in `/usr/local/mysql/lib` or on the shared library path. This can be overridden by setting the special variable `*mysql-library-directories*`.

By default, LispWorks expects the library to be called `libmysqlclient.*.dylib` and it searches for a library that matches that pattern, where `*` is any version number. This search can be avoided by setting `*mysql-library-path*` to something other than the default `"-lmysqlclient"`, for example, it is possible to force LispWorks to look for version 12 by evaluating

```lisp
(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*`.

19.2.9 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.

19.2.9.1 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/7.4/static/libpq.html#LIBPQ-CONNECT
```
For example,

```
dbname=test user=scott password=tiger host=scandium
```

### 19.2.9.2 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 `:port` can be an integer or a string naming a service.

If `:connection` is specified then it can 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.

### 19.3 Functional interface

The functional interface provides a full set of Data Manipulation and Data Definition functions. The interface provides an 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.
19.3 Functional Data Manipulation Language (FDML)

The functions available for Data Manipulation and Data Definition are described below.

19.3.1 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 19.5 on page 243. For example, the following is a potential query and its result:

```
(select [person_id] [person surname] :from [person])
=>
((111 "Brown") (222 "Jones") (333 "Smith"))
("PERSON_ID" "SURNAME")
```

In this example, [person_id], [person surname] and [person] are database-identifiers and evaluate to literal SQL. The result is a list of lists of attribute values. Conversely, consider

```
(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:
(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 256.

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"))

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>22000</td>
</tr>
<tr>
<td>Jones</td>
<td>45000</td>
</tr>
<tr>
<td>Smith</td>
<td>35000</td>
</tr>
</tbody>
</table>
19.3 Functional interface

19.3.1.2 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:

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

and:

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

are equivalent to the following SQL:

```
INSERT INTO PERSON
    (PERSON_ID, INCOME, Surname, OCCUPATION)
VALUES (115, 11000, 'Johnson', 'plumber')
```

If query is provided, then neither values nor av-pairs should be. In this case the attribute names in the query expression must also exist in the insertion table. For example:
(insert-records :into [person]
  :query [select [id] [firstname] [surname]
  :from [manager]]
  :attributes '((person_id firstname surname))
)

To delete or alter those records in a table which match some condition, use delete-records or update-records.

19.3.1.3 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*.

19.3.1.4 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 com-
mit 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.

```lisp
(connect "personnel")
(with-transacation
  (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`.

### 19.3.1.5 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
```
The more general word *tuple* is used so that it can also be applied to the object-oriented case. In the functional case, *tuple* is synonymous with *record*.

Each iteration of the loop assigns the next record of the table to the variable *var*. The record is represented in Lisp as a list. Destructuring can be used in *var* to bind variables to specific attributes of the records resulting from *query-expression*. In conjunction with the panoply of existing clauses available from the *loop* macro, the new iteration clause provides an integrated report generation facility.

Suppose the name of everyone in an employee table is required. This simple query is shown below using the different iteration method. The function *map-query* requires *flatp* to be specified; otherwise each name would be wrapped in a list.

```
(do-query ((name) [select [ename] :from [emp]])
  (print name))

(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))
```
19.3 Functional interface

19.3.1.6 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 string is known in advance and thus the square bracket syntax is not needed.

The function `query` runs a SQL query on a database and returns a list of values like `select` (see “Querying” on page 231). 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 946.

19.3.1.7 Building vendor-specific SQL

Common SQL does not provide a general interface to vendor-specific syntax. There are two approaches you can take with SQL such as this:

```sql
SELECT B.PARTY_CODE_ALIAS, A.VALUE FROM CODES A, CODE_ALIASES B
WHERE A.DOMAIN=B.CODE_DOMAIN(+) AND A.VALUE=B.CODE_VALUE(+) AND B.PARTY_ID(+)=<party_id>
```

1. Construct the string as above and then call `query` as described in “Specifying SQL directly” on page 237.

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

```sql
{sql:select [b party_code_alias] [a value]
 :from '([codes a] [codes_aliases b])
 :where [and [= [a domain]
    {sql:sql-expression
     :string "B.CODE_DOMAIN(+)"
    }]
    [= (sql:sql-expression
     :string "B.PARTY_ID(+)" PARTY-ID])

19.3.2 Functional Data Definition Language (FDDL)

Functions in the FDDL may be used to change or query the structure of the database.
19.3.2.1 Querying the schema

The functions list-tables, list-attributes, attribute-type and list-attribute-types return information about the structure of a database.

19.3.2.2 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
```

19.3.2.3 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.

19.4 Object oriented interface

This section describes the object-oriented interface to SQL databases using specialized CLOS classes. These classes have standard-db-object as one of their superclasses and have a common metaclass which provides the specialized behavior for mapping subclasses of standard-db-object onto records in the database. A class of this kind is created using def-view-class.

19.4.1 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
19.4 Object oriented interface

- 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.

19.4.1.1 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).

19.4.2 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.
19.4.2.1 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 930 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.

19.4.3 Object-Oriented Data Manipulation Language (OODML)

The OODML is designed to be powerful and expressive, while remaining familiar to users of the FDML. To achieve this aim, some of the functions and macros in the SQL interface have been overloaded — particularly the `select` function and the iteration constructs.

The function `select` is common across the both the functional and object-oriented SQL interfaces. If its first argument, `selections`, refers to a View Class by supplying its symbolic name then the select operation becomes object-oriented and it returns a list of instances instead of a list of attributes.

A subsequent equivalent `select` call will return the same (eq) instances. The `:refresh` argument can be used to ensure that existing instances get updated with any changed data. If such an update requires action by your application, then add methods on the generic function `instance-refreshed`.

In a View Class `select` call, the symbol `slot-value` is a valid SQL operator for use within the `:where` argument.

To find the View Classes for a particular database, use the function `list-classes`.

To manipulate data via a View Class, that is to modify the records corresponding to instances of the View Class, using the generic functions `update-records-from-instance`, and `update-record-from-slot`.

To delete records corresponding to instances of the View Class, use the generic function `delete-instance-records`.

To update existing instances of a View Class when data is known to have changed, use the generic functions `update-slot-from-record` and `update-instance-from-records`. 
19.4.3.1 Examples

[select 'employee]  
  -> #<SQL-OBJECT-QUERY (EMPLOYEE)>

(select 'employee  
  :where [= [slot-value 'employee 'employee-job]  
           "SALESMA"]
  ((#<db-instance EMPLOYEE 8067092>)  
   (#<db-instance EMPLOYEE 8069536>)  
   (#<db-instance EMPLOYEE 8069176>))

(list-classes)  
(#<db-class EMPLOYEE> #<db-class DEPARTMENT>)

19.4.3.2 Iteration

The object-oriented SQL interface has the same three iteration constructs as  
the functional interface (see Section 19.3.1.5 on page 235): 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 'organisation  
    :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.

19.4.3.3 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.
19.5 Symbolic SQL syntax

Common SQL supports a symbolic query syntax across both the functional and object-oriented interface layers. It allows SQL and Common Lisp expressions to be mixed together — with as much processing as possible done at compile-time. Symbolic SQL expressions are read as square-bracketed lists to distinguish them from Lisp expressions. However, each can be nested within the other to achieve the desired result.

By default, this reader syntax is turned off. To turn it on see Section 19.5.3 on page 249.

19.5.1 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 an SQL string representing a symbolic expression
3. To enclose literal SQL

Each of these uses is demonstrated below.

19.5.1.1 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
```
This corresponds to a qualified SQL identifier with an aliased table name containing special characters as in:

```
SELECT "foo".BAR FROM BAZ "foo".
```

This corresponds to an alias definition as in:

```
SELECT "bar".* FROM FOO "bar".
```

As above, but including a type coercion component.

As above, but includes a type coercion component.

As above, but includes a type coercion component.

### 19.5.1.2 SQL strings representing symbolic expressions

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

```
[any '3 4] -> #<SQL-VALUE-EXP "ANY (3,4)"
```

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

```
[> [baz] [beep]]
-> #<SQL-RELATIONAL-EXP "(BAZ > BEEP)"
```

The `select` statement itself may be prepared for later query execution using the `[:from]` syntax. For example:

```
[select [person_id] [surname] :from [person]]
```

This form results in an SQL expression, which could be bound to a Lisp variable and later given to `query` to execute. For example:
19.5 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 an 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 247).
The general syntax is: `<operator> <operand> ...`, for instance:

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

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

```sql
(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]]])
```
19.5 Symbolic SQL syntax

19.5.1.3 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])
```

Also you can call SQL infix operators using the pseudo operators sql-boolean-operator and sql-operator.

19.5.1.4 Enclosing literal SQL

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

```sql
["SELECT FOO, BAR FROM BAZ"]
```

19.5.2 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 1019.
The function `sql-expression` makes an SQL expression from the given key-words. This is equivalent to the first and third uses of the `[]` syntax as discussed in Section 19.5.1 on page 243.

The function `sql-operator` returns the Lisp symbol for an 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 an SQL string constant). `nil` maps to "NULL", that is, an 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 1014.

19.5.2.1 Examples

The following example function, taken from the object-oriented SQL interface layer, makes an 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 an SQL `select` statement:
19.6 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)
```

19.6 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 256 for information specifically about returning datetime values from MySQL.

19.6.1 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] "25-Dec-2005"])
```

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.

19.6.2 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])
```

19.6.2.1 Timezone of returned DATES

Common SQL creates universal time values from DATE fields assuming that the database contains times in Coordinated Universal Time (UTC). That is, as if by passing `time-zone` 0 to `encode-universal-time`. To decode the values consistently with this encoding, pass `time-zone` 0 to `decode-universal-time`.

If the database contains times in a different timezone, then the integer `time-zone` needs to be adjusted by adding an appropriate multiple of 3600 before calling `decode-universal-time`. 
19.6.3 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: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 1008 for details.

19.6.4 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.

19.7 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`.

19.8 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.
19.8.1 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 signalled with the wrong condition class, please report them to Lisp Support, including the full printout of the condition, and we will fix it.

19.8.2 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.
19.9 Using MySQL

The section describes particular issues in Common SQL with MySQL databases.

19.9.1 Connection specification

See “Connecting to MySQL” on page 227 for information about MySQL specific extensions for the connection-spec passed to connect.

19.9.2 Case of table names and and database names

MySQL is case sensitive on table names and database names when the server is on a Unix machine. MySQL does not automatically change raw names to uppercase as specified by the SQL standard. However, Common SQL is geared towards uppercasing all names, so this may cause some mismatches. In general, Common SQL uppercases strings, and uses symbol names, which are normally uppercase, as-is.

One solution, possible only if you control the naming of tables and databases, is to make them all have the same case. If this is uppercase, that suffices. If it is lowercase, you need to set the variable lower_case_table_names in the configuration of the server.

If you cannot make all the names the same case, you have to get the case right. This can be achieved in several ways:

1. Specify tables names using strings, for example:
   
   (sql:select [*] :from "TableNAMEwithVARIABLEcase")
   
   Note that this does not work in LispWorks 4.4 and previous versions.

2. Pass the Lisp string directly:
   
   (sql:select [*] :from "TableNAMEwithVARIABLEcase")

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 1016.

sql-error-database-message is a string (maybe nil) returned by the foreign code.
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 254 for details).

3. Specify table names as escaped symbols:

\[ \text{sql:select } [*] \text{ :from } \{\text{TableNAMEwithVARIABLEcase}\}\]

4. Construct the whole query string and pass it to `query` rather than using `select`:

\[ \text{sql:query } "\text{select } * \text{ from TableNAMEwithVARIABLEcase}" \]

### 19.9.3 Encoding (character sets in MySQL).

You can specify the encoding to be used by passing the `:encoding` argument to `connect`. Common SQL supports various encodings for MySQL as documented in `connect`, page 916.

The default is to use the default for the particular MySQL installation.

### 19.9.4 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:

\[ \text{sql:connect } "\text{me/mypassword/mydb}" \; \text{:sql-mode } "\text{ANSI_QUOTES}" \]

See the description of the `:sql-mode` argument to `connect`, page 916 for details.

### 19.9.5 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.
19.9  Using MySQL

19.9.6  Special considerations for iteration functions and macros

This section describes particular issues when fetching multiple records using Common SQL with MySQL databases.

19.9.6.1  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 forname,surname from people"
    :get-all t)
```

```lisp
(sql:do-query
 (forname surname) "select forname,surname from people"
 :get-all t)
```

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

```lisp
(loop for (forname surname) being each record
    "select forname,surname from people"
    :get-all t
    body)
```

19.9.6.2  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.

### 19.9.7 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:

```
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`.

### 19.9.8 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`.

### 19.9.9 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 19.2

<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 231.

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.

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

19.10 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 259.

19.10.1 Connection specification
See “Connecting to Oracle” on page 225 for information about Oracle-specific interpretation of the connection-spec passed to connect.

19.10.2 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).
19.11 Oracle LOB interface

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.

19.11 Oracle LOB interface

19.11.1 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.

19.11.1.1 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.

19.11.1.2 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 260 for details.
19.11.1.3 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 be called by using the pseudo operator sql-function, like this:

\[
\text{\begin{verbatim}
(sql:insert-records :into [mytable]
 :values
 (list "name" [sql-function 'empty_blob]))
\end{verbatim}
}\]

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.

19.11.2 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:

\[
\text{\begin{verbatim}
create table mytable {
 name varchar(200),
 image  blob
}
\end{verbatim}
}\]

Then doing

\[
\text{\begin{verbatim}
(sql:select [image] :from [mytable] :flatp t)
\end{verbatim}
}\]

returns a list of LOB locators.

This example lists the size of the images in the table mytable:

\[
\text{\begin{verbatim}
(dolist (pair (sql:select [name][image] :from [mytable]))
 (format t "\%-a has an image of size \%-a\n"
 (first pair) (sql:ora-lob-get-length (second pair)))
(sql:ora-lob-free (second pair)))
\end{verbatim}
}\]

or more efficiently

\[
\text{\begin{verbatim}
(sql:do-query ((name lob-locator)
 [sql:select [name][image] :from [mytable]])
 (format t "\%-a has an image of size \%-a\n"
 name (sql:ora-lob-get-length lob-locator)))
\end{verbatim}
}\]
19.11 Oracle LOB interface

**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*.

### 19.11.3 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 1008 for details. For example, writing a line in the end of the log file of station number 573:

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

(sql:do-query ((log-stream)
  [select [log :output-stream] :from [logfiles]
(file-position log-stream :end)
(write-line "Add this line to the log" log-stream)
(close log-stream) ; forces the output
)

(sql:commit)
```

Note that any call to *commit* or *rollback* on the same connection removes the lock. If you want to modify the LOB later, you must lock it again. An efficient way to achieve this is to use the special token ROWID, which returns the ROWID in the database, because this does not involve searching on the server side. For example:
(let ((lobs-list
  (sql:select [lob-field] [rowid] ; get pairs of LOB
    :from [mytable] ; locators and ROWIDs
    :where [some-condition]))))

... do something ...

... reach a point when we want to modify one of the LOBS above and have bound one of the pairs in the variable pair.

(sql:select [1]
  :from [mytable] ; retrieve a constant
  :where
    [= [rowid] (second pair)] ; get the right record
    :for-update t) ; lock it

(sql:ora-lob-write-buffer (car pair) ; modify the lob
  offset
  amount
  buffer)

(sql:commit) ; also unlock everything
)

19.11.4 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.
19.11.5 Attaching a stream to a LOB locator

It is possible to attach a stream to a LOB locator, passing the LOB locator as a
\texttt{:lob-locator} argument to (\texttt{make-instance 'lob-stream ...}). The value of
the \texttt{:direction} argument must be \texttt{:input} or \texttt{:output}. By default, if the stream
is closed the LOB locator is freed, unless the value of the initarg \texttt{:free-lob-locator-on-close} is passed as \texttt{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 \texttt{force-output}
or \texttt{clear-input} before accessing the LOB in other ways to avoid these prob-
lems.

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 \texttt{: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 \texttt{lob-stream-lob-locator}.

19.11.6 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 \texttt{ora-lob-lob-locator}, and maybe \texttt{ora-lob-env-handle} and \texttt{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 \texttt{ora-lob-read-foreign-buffer}, \texttt{ora-lob-write-
foreign-buffer} and \texttt{ora-lob-get-buffer}.

For example:
You have a C function `my_lob_processor`

```lisp
(fli:define-foreign-function my-lob-processor
  ((lob sql:p-oci-lob-locator)
   (env sql:p-oci-svc-ctx)
   (other-arg :int))
  :result-type :int)
```

Assuming you have the LOB locator in the variable `lob`, call the foreign function on it:

```lisp
(my-lob-processor (sql:ora-lob-lob lob)
                   (sql:ora-lob-svc-ctx-handle lob) 36)
```

There are three handles in the LOB: the LOB descriptor itself, the environment and the context. The pointer types, the reader and the corresponding C type for each handle are shown in Table 19.3 below.

<table>
<thead>
<tr>
<th>OCI handle</th>
<th>Reader</th>
<th>Pointer type</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOB descriptor</td>
<td>ora-lob-lob-locator</td>
<td>p-oci-lob-locator</td>
<td>OCILOBLocator*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or p-oci-file</td>
<td></td>
</tr>
<tr>
<td>context</td>
<td>ora-lob-svc-ctx-handle</td>
<td>p-oci-svc-ctx</td>
<td>OCISvcCtx*</td>
</tr>
<tr>
<td>environment</td>
<td>ora-lob-env-handle</td>
<td>p-oci-env</td>
<td>OCIEnv*</td>
</tr>
</tbody>
</table>

The `p-oci-lob-locator` pointer type is used for internal LOBs (that is, BLOB, CLOB and NCLOB). The `p-oci-file` pointer type 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.
19.11 Oracle LOB interface

19.11.7 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 `simple-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.

19.11.8 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 262. 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 424. 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.

19.11.9 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.

19.11.9.1 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 265.
`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 263. 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`.

19.11.9.2 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`.

19.11.9.3 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 261.

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.

### 19.11.9.4 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`.

### 19.11.9.5 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.

19.11.9.6 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.

19.11.9.7 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.
20

User Defined Streams

20.1 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 27, “The COMMON-LISP Package” for alterations to Common Lisp functions, and Chapter 39, “The STREAM Package” for more details on the API for user defined streams.

20.2 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.
20.2.1 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.

20.2.2 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 `simple-char`. The following defines a method on `stream-element-type` to return the correct element type.

```
(defmethod stream-element-type ((stream unicode-ls-stream))
  'simple-char)
```

20.2.3 Stream directionality

Streams can be defined for input only, output only, or both. In our example, the `unicode-ls-stream` class needs to be able to read from a file and write to a file, and we therefore defined it to inherit from an input and an output stream class. We could have defined disjoint classes instead, one inheriting from `fun-
20.2 An illustrative example of user defined streams

damental-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.

20.2.4 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.

```
(defmethod stream:stream-read-char ((stream unicode-ls-stream))
  (let ((char (read-char (ls-stream-file-stream stream) nil :eof)))
    (if (eq 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 stream-listen and stream-clear-input would work for our stream, it is faster to use the functions provided by file-stream, again using our accessor ls-stream-file-stream.

(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)))

### 20.2.5 Stream output

The following method for stream-write-char uses write-char to write a character to the stream. If the character written to unicode-ls-stream is a \#\Newline, then the method writes a \#\Line-Separator to the file stream.

(defmethod stream:stream-write-char ((stream unicode-ls-stream) char)
  (write-char (if (eq char \#Newline)
                \#Line-Separator
                char)
             (ls-stream-file-stream stream)))

The default method for stream-write-string calls stream-write-char repeatedly to write a string to the stream. However, the following is a more efficient implementation for our stream.
20.2 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)))
20 User Defined Streams

(defmethod stream:stream-clear-output ((stream
  unicode-ls-stream))
  (clear-output (ls-stream-file-stream stream)))

20.2.6 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 'simple-char)))

The following macro uses open-unicode-ls-stream in a similar manner to the
Common Lisp macro with-open-file:

(defun with-open-unicode-ls-file ((var filename
  &key (direction :input))
  &body body)
  `(let ((,var (open-unicode-ls-file ,filename
    :direction ,direction)))
      (unwind-protect
        (progn ,body)
        (close ,var)))

We now have the required functions and macros to test our user defined
stream. The following code uses config.sys as a source of input to an
instance of our stream, and outputs it to the file unicode-ls.out, changing all
occurrences of \
Newline to \Line-Separator in the process.

(with-open-unicode-ls-file (ss "C:\unicode-ls.out"
  :direction :output)
  (write-line "-*- Encoding: Unicode; -*" ss)
  (with-open-file (ii "C:\config.sys") ; Don't edit this file!
    (loop with line = nil
      while (setf line (read-line ii nil nil))
      do (write-line line ss))))
After running the above code, if you load the file C:\unicode-1s.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 following code to print out the contents of the new file line by line.

```lisp
(with-open-unicode-ls-file (ss "C:\unicode-1s.out")
  (loop while (when-let (line (read-line ss nil nil))
    (write-line line)))))
```
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`.

**Note:** to load the Socket Stream SSL interface, evaluate

```
(require "comm")
```

**Note:** Below we assume that the current package uses the `comm` package. That is, `comm` package symbols may not be qualified explicitly.

### 21.1 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)
```
21 Socket Stream SSL interface

21.2 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.

21.3 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.

21.3.1 OpenSSL constants

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

21.3.2 Naming conventions for direct OpenSSL calls

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

21.3.2.1 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.
21.3.2.2 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"

21.3.3 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 \texttt{ssl-pointer}, \texttt{ssl-ctx-pointer} or \texttt{ssl-cipher-pointer}. Where an OpenSSL function takes a \texttt{char*} or \texttt{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 21.1 Direct calls to OpenSSL

<table>
<thead>
<tr>
<th>Lisp function</th>
<th>Return values</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ssl-add-client-ca}</td>
<td></td>
</tr>
<tr>
<td>\texttt{ssl-cipher-get-bits}</td>
<td>First value is number of bits the cipher actually uses.</td>
</tr>
<tr>
<td></td>
<td>Second value is number of bits the algorithm of the cipher can use (which may be higher).</td>
</tr>
</tbody>
</table>
Table 21.1 Direct calls to OpenSSL

<table>
<thead>
<tr>
<th>Lisp function</th>
<th>Return values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssl-cipher-get-name</td>
<td>string.</td>
</tr>
<tr>
<td></td>
<td>e.g. &quot;DHE-RSA-AES256-SHA&quot;</td>
</tr>
<tr>
<td>ssl-cipher-get-version</td>
<td>string.</td>
</tr>
<tr>
<td></td>
<td>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>
<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>
</tbody>
</table>
### Table 21.1 Direct calls to OpenSSL

<table>
<thead>
<tr>
<th>Lisp function</th>
<th>Return values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ssl-ctx-set-read-ahead</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-ctx-set-tmp-rsa</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-ctx-set-tmp-dh</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-ctx-use-certificate-chain-file</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-ctx-use-certificate-file</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-ctx-use-privatekey-file</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-ctx-use-rsaprivatekey-file</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-get-current-cipher</code></td>
<td><code>ssl-cipher-pointer</code></td>
</tr>
<tr>
<td></td>
<td>Can be a null pointer.</td>
</tr>
<tr>
<td><code>ssl-get-max-cert-list</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-get-mode</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-get-options</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-get-verify-mode</code></td>
<td><code>integer</code></td>
</tr>
<tr>
<td><code>ssl-get-version</code></td>
<td><code>string</code></td>
</tr>
<tr>
<td></td>
<td>&quot;TLSv1&quot;, &quot;SSLv2&quot; or &quot;SSLv3&quot;</td>
</tr>
<tr>
<td><code>ssl-load-client-ca-file</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-need-tmp-rsa</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-num-renegotiations</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-session-reused</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-set-accept-state</code></td>
<td><code>None</code></td>
</tr>
<tr>
<td><code>ssl-set-client-ca-list</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-set-connect-state</code></td>
<td><code>None</code></td>
</tr>
<tr>
<td><code>ssl-set-max-cert-list</code></td>
<td></td>
</tr>
<tr>
<td><code>ssl-set-mode</code></td>
<td></td>
</tr>
</tbody>
</table>
If you need OpenSSL functionality that is not provided here, you can define your own foreign functions via the LispWorks Foreign Language Interface. If you do this, an important point to note is that on Microsoft Windows, the \texttt{:calling-convention} must be \texttt{:cdecl} (it defaults to \texttt{:stdcall}). If using OpenSSL suddenly causes mysterious crashes, the \texttt{:calling-convention} in your foreign function definitions is the first thing to check.

### 21.4 Socket Stream SSL keyword arguments

The keyword arguments \texttt{:ssl-ctx}, \texttt{:ssl-side}, \texttt{:ctx-configure-callback} and \texttt{:ssl-configure-callback} can be 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 21.2, page 283.

<table>
<thead>
<tr>
<th>Lisp function</th>
<th>Return values</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ssl-set-options}</td>
<td></td>
</tr>
<tr>
<td>\texttt{ssl-set-temp-rsa}</td>
<td></td>
</tr>
<tr>
<td>\texttt{ssl-set-temp-dh}</td>
<td></td>
</tr>
<tr>
<td>\texttt{ssl-total-renegotiations}</td>
<td></td>
</tr>
<tr>
<td>\texttt{ssl-use-certificate-file}</td>
<td></td>
</tr>
<tr>
<td>\texttt{ssl-use-rsaprivatekey-file}</td>
<td></td>
</tr>
<tr>
<td>\texttt{ssl-use-privatekey-file}</td>
<td></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 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`
- `:server` => `*_server_method`
- `:both` => `*_method`

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 not a ssl-pointer, ssl-configure-callback is ignored.

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

You can attach SSL to an existing socket-stream by calling attach-ssl on the stream. 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 attach-ssl as described in “Socket Stream SSL keyword arguments” on page 282.

Detach SSL from a socket-stream and shut down the SSL with detach-ssl. For full descriptions see attach-ssl, page 345 and detach-ssl, page 348.

### 21.6 Using SSL objects directly

The C objects SSL and SSL_CTX are represented in LispWorks by foreign pointers with type ssl-pointer and ssl-ctx-pointer, which correspond to the C types SSL* and SSL_CTX*. These foreign types should be used for any foreign function that takes or returns these C types, and must be used when passing a foreign pointer as the value of the :ssl-ctx argument.

Making SSL objects is a way of getting access to them to perform configuration, but, especially in the case of the SSL_CTX, it is a useful way to avoid repeated calls to the configuration routines which may be time consuming. For example, if we have defined a function configure-a-ctx, and we want to read once every 60 seconds from some URL, we can write:

```lisp
(loop (with-open-stream
   (str (comm:open-tcp-stream some-url 443 :ssl-ctx t
      :ctx-configure-callback
      'configure-a-ctx))
   (read-something str))
   (sleep 60))
```
This will cause `configure-a-ctx` to be called each time. If it is expensive, we can call it only once by changing the code to:

```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.

### 21.7 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.

### 21.8 Obtaining and installing the OpenSSL library

At the time of writing, OpenSSL is available as shown in Table 21.3:

<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.siproweb.com/products/Win32OpenSSL.html">www.siproweb.com/products/Win32OpenSSL.html</a></td>
</tr>
</tbody>
</table>
21.8 Obtaining and installing the OpenSSL library

21.8.1 Installing the OpenSSL library on Solaris

After installing (with pkgadd) you need to put the shared libraries `libcrypto.so` and `libssl.so` 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`.

21.8.2 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 21.4, page 287.

```
<table>
<thead>
<tr>
<th>Operating System</th>
<th>Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>-lssl</td>
</tr>
<tr>
<td>Windows</td>
<td>libeay32.dll libssl32.dll</td>
</tr>
</tbody>
</table>
```

Table 21.3 OpenSSL availability

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Availability of OpenSSL</th>
</tr>
</thead>
<tbody>
<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>x86/x64 Solaris</td>
<td>Installed by default</td>
</tr>
<tr>
<td>SPARC Solaris</td>
<td>Installed by default on Solaris 10. For other versions, see the freeware from Sun at <a href="http://sunfreeware.com">sunfreeware.com</a> for both 32-bit and 64-bit.</td>
</tr>
</tbody>
</table>
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`.

### 21.9 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.
22

Internationalization

22.1 Introduction
LispWorks uses Unicode (UCS-2 encoding) internally in its representation of character objects. All Unicode characters can be represented in strings, though 8-bit string types are also provided for efficiency when characters beyond the Latin-1 range are not needed. Character and string data can be input and output in various encodings (external formats).

22.2 Character and String types

22.2.1 Character types
The following subtypes of character are defined:

- **base-char** simple characters with char-code less than base-char-code-limit (256).
- **simple-char** simple characters with char-code less than char-code-limit (65536).
- **character** All characters including non-simple characters (that is, with non-null bits attributes).
22.2.2 Character Syntax

All simple characters have names that consist of \texttt{U+} followed by the code of the character in hexadecimal, for example \texttt{#\U764F} is \texttt{(code-char #x764F)}.

Additionally, Latin-1 characters have names derived from the ISO10646 name, for example:

\begin{verbatim}
(char-name (code-char 190)) => "Vulgar-Fraction-Three-Quarters"
\end{verbatim}

Names are also provided for space characters:

\begin{verbatim}
(name-char "Ideographic-Space") => #\Ideographic-Space
\end{verbatim}

If required, the bits attributes names can be prepended as usual:

\begin{verbatim}
#\ctrl-ideographic-space => #\Control-Ideographic-Space
\end{verbatim}

22.2.3 String types

String types are supplied which are capable of holding each of the character types mentioned above. The following string types are defined:

- \texttt{base-string} holds any \texttt{base-char}
- \texttt{text-string} holds any \texttt{simple-char}
- \texttt{augmented-string} holds any \texttt{character}.

In particular, \texttt{text-string} is the type that can hold all characters used in texts. The types above include non-simple strings - those which are displaced, adjustable or with a fill-pointer.

The Common Lisp type \texttt{string} itself is dependent on the value of \texttt{*default-character-element-type*} according to the rules for string construction described in “String Construction” on page 292. For example:
22.2  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: In a call to SEQ:%SET-ACCESS-ARRAY: \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, :c <option number> to proceed, or :? for other options
CL-USER 3 : 1 > :a

CL-USER 4 > (set-default-character-element-type 'simple-char)
SIMPLE-CHAR

CL-USER 5 > (coerce (list \Ideographic-Space) 'string)
"\n
The following types are subtypes of simple-string. Note that in the names of the string types, ’simple’ refers to the string object and does not mean that the string’s elements are simple-chars.

- simple-base-string holds any base-char
- simple-text-string holds any simple-char
- simple-augmented-string holds any 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 292.

22.2.3.1 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)
ambiguous because it needs to select a concrete type (such as `simple-base-string` or `simple-text-string`).

When LispWorks is running with `*default-character-element-type*` set to `base-char`, it expects that you will want strings with element type `base-char`, so functions like `coerce` treat references to `simple-string` as if they were `(simple-array base-char *)`.

If you call `set-default-character-element-type` with a larger character type, then `simple-string` becomes a union of the array types that are subtypes of that character type.

### 22.2.3.2 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-text-string` or `simple-augmented-string`.

### 22.3 String accessors

`schar` works on any simple string object. However, for efficient string access when a simple string type is known, the following specialised accessors are provided:

- `sbchar` for `simple-base-string`.
- `stchar` for `simple-text-string`.

### 22.4 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*`. 
22.4.1 Default string construction

If the value of \texttt{*default-character-element-type*} is \texttt{base-char} then:

\begin{verbatim}
(make-string 3)
\end{verbatim}

returns a \texttt{simple-base-string} and

\begin{verbatim}
(coerce sequence 'simple-string)
\end{verbatim}

attempts to construct a \texttt{simple-base-string}. This will signal an error if any element of \texttt{sequence} is not a \texttt{base-char}.

If the value of \texttt{*default-character-element-type*} is \texttt{simple-char} then

\begin{verbatim}
(make-string 3)
\end{verbatim}

returns a \texttt{simple-text-string} and

\begin{verbatim}
(coerce sequence 'simple-string)
\end{verbatim}

attempts to construct a \texttt{simple-text-string}. This will signal an error if any element of \texttt{sequence} is not a \texttt{simple-char}.

Other string constructors also take their default from \texttt{*default-character-element-type*}. For instance, the string reader will always construct a string of type determined by this variable, unless it sees a character of a larger type, in which case a suitable string is constructed. Also \texttt{with-output-to-string} and \texttt{make-string-output-stream} will construct a stream with element type determined by this variable and generate a string of the same element type.

22.4.2 String construction with known type

The variable \texttt{*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:

\begin{verbatim}
(make-string 3 :initial-element #'\Ideographic-Space)
\end{verbatim}

constructs a string of a type that can hold its elements, regardless of the value of \texttt{*default-character-element-type*}. 
22.4.3 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 specialised 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 'simple-char)

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*.

22.4.4 String construction on Windows systems

When LispWorks for Windows starts up on a OS with a non-Latin-1 code page, it calls

(set-default-character-element-type 'simple-char)

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 simple-char.
22.5 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**
  The UCS-2 encoding of Unicode. The parameter :little-endian defaults to the endianness of the platform.

- **:utf-8**
  The UTF-8 encoding of Unicode.

- **:jis**
  JIS. The encoding data is read from a file Uni2JIS and is pre-built into LispWorks.

- **:euc-jp**
  EUC-JP. The encoding data is read from a file Uni2JIS and is pre-built into LispWorks.

- **:sjis**
  Shift JIS.
22 Internationalization

:windows-cp936 Windows code page 936. The encoding data is read from a file windows-936-2000.ucm and is pre-built into LispWorks.

:gbk A synonym for :windows-cp936.

Note: windows-936-2000.ucm is provided by way of documentation in the directory lib/6-0-0-0/etc/. It is not read at runtime.

Note: Uni2JIS is provided by way of documentation in the directory lib/6-0-0-0/etc/. It is also used at runtime by the function char-name.

22.6 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.

22.6.1 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.

22.6.2 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:
22.6 External Formats and File Streams

(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.

22.6.3 Guessing the external format

If open or with-open-file gets a non-complete :external-format argument ef-spec then the system decides which external format to use by calling the function guess-external-format.

The default behavior of guess-external-format is as follows:

1. When ef-spec’s name is :default, this finds a match based on the file-name; or (if that fails), looks in the Emacs-style (-*-) attribute line for an option called ENCODING or EXTERNAL-FORMAT; or (if that fails), chooses from amongst likely encodings by analysing the bytes near the start of the file, or (if that fails) uses a default encoding. Otherwise ef-spec’s name is assumed to name an encoding and this encoding is used.

2. When ef-spec does not include the :eol-style parameter, it then also 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 (UCS-2) encoded. The routine in step 1 above detects this:

(set-default-character-element-type 'simple-char)
=> SIMPLE-CHAR

(with-open-file (ss "C:/temp/unicode-notepad.txt")
  (stream-external-format ss))
=> (:UNICODE :LITTLE-ENDIAN T :EOL-STYLE :CRLF)
The behavior of `guess-external-format` is configurable via the variables
`*file-encoding-detection-algorithm*` and `*file-eol-style-detection-algorithm*`. See the manual pages for details.

### 22.6.3.1 Example of using UTF-8 by default

To change the default for all file access via `open`, `compile-file` and so on, you can modify the value of `*file-encoding-detection-algorithm*`.

For example given the following definition:

```lisp
(defun utf-8-file-encoding (pathname ef-spec buffer length)
  (declare (ignore pathname buffer length))
  (system:merge-ef-specs ef-spec :utf-8))
```

then this makes it use UTF-8 as a fallback:

```lisp
(setq system:*file-encoding-detection-algorithm*
  (substitute 'utf-8-file-encoding
               'system:locale-file-encoding
               system:*file-encoding-detection-algorithm*))
```

and this forces it to always use UTF-8:

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

### 22.6.4 External formats and stream-element-type

The `:element-type` argument in `open` and `with-open-file` defaults to the value of `*default-character-element-type*`.

If `element-type` is not `:default`, checks are made to ensure that the resulting stream’s `stream-element-type` is compatible with its external format:

1. If `direction` is `:input` or `:io`, the `element-type` argument must be a supertype of the type of characters produced by the external format.
2. If `direction` is `:output` or `:io`, the `element-type` argument must be a subtype of the type of characters accepted by the external format.

If the `element-type` argument doesn’t satisfy these requirements, an error is signalled.
If `element-type` is `:default` the system chooses the `stream-element-type` on the basis of the external format.

### 22.6.5 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 (UCS-2) 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.

### 22.6.6 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 Unicode (UCS-2 encoded, `external-format :unicode`) file regardless of whether the BOM is present.

Some editors including Microsoft Notepad and the LispWorks editor write the BOM when writing a file with Unicode (UCS-2) encoding.

### 22.7 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*.

### 22.8 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.
22.8.1 Unicode case insensitive character comparison


22.8.2 Unicode case insensitive string comparison


22.8.3 Unicode character predicates

This chapter describes the interfaces which provide information about the environment in which LispWorks is running. This includes the operating system, the physical location of the LispWorks executable, and the arguments it was passed on startup.

### 23.1 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 Millennium Edition, Windows 2000, Windows XP, Windows Vista and so on. See the manual pages for details.

### 23.2 Site Name

The Common Lisp functions `short-site-name` and `long-site-name` can be configured using `setf:`
(setf (long-site-name) "LispWorks Ltd"
     (short-site-name) "LW")

23.3 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.

23.4 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=)

23.4.1 Command Line Arguments

The following command line options are supported by the system.

-build build-script

build-script names a file to be loaded on startup, typically for the purpose of building another image. LispWorks quits after loading the file. If an error is signalled while loading the file, a backtrace is displayed and LispWorks quits.
An image run with `-build` runs itself, and not the default saved session if you created one. See “Saved sessions” on page 133 for information on saved sessions.

**Note:** `init-file` and `siteinit-file` are not loaded when using `-build`, so your build script file must call `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.

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

- **-IIOPnumeric**
  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 130.
Your default initialization file can be set in the LispWorks IDE. See “Setting global preferences” in the LispWorks IDE User Guide for details.

If init-file is not found, an error is signalled. 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 133 for information on saved sessions.

-multiprocessing
Initializes multiprocessing on startup. See Chapter 15, “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 605.

-ORBport orbport
orbport specifies a port number for the LispWorks ORB. The special value 0 allows the system to pick a port.

--relocate-image BaseAddress
Causes the image to relocate at BaseAddress on supported platforms, as described in “Startup relocation” on page 306. 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 6.0.
23.5 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 306.

23.5.1 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.

23.5.2 Layout of memory

This is Operating System-dependent:

On Solaris, 32-bit LispWorks is mapped at \texttt{#x10000000}. In principle it can grow to almost \texttt{#x80000000} (the libraries are at higher addresses).

On HP-UX, 32-bit LispWorks is mapped at \texttt{#x50000000}, because it cannot use the first quadrant. The libraries are also mapped at the same quadrant, at around \texttt{#x7a000000}, so the total size can be a little more than 0.5GB.
For the other platforms and for 64-bit LispWorks, see the discussion in “Start-up relocation” on page 306.

### 23.5.3 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`.

### 23.6 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 supported platforms. The discussion in this section is applicable to all relocatable implementations.

On Microsoft Windows and Macintosh, LispWorks detects memory clashes and avoids them automatically. On these platforms there is no need to explicitly relocate LispWorks. The other relocatable implementations - LispWorks (32-bit) for Linux, LispWorks (64-bit) for Linux, LispWorks (32-bit) for FreeBSD, LispWorks (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.

### 23.6.1 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:
23.6 Startup relocation

--relocate-image BaseAddress

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

--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`.

23.6.2 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 growing 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.

### 23.6.2.1 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.

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 6.0.

### 23.6.2.2 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.

If the memory above LispWorks gets used by other parts of the software after LispWorks was mapped, it may be possible to avoid the problem by reserving some memory above LispWorks by using ReserveSize.

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

In principle LispWorks can grow up to some distance below #xC000000 (almost 2.25GB), though this depends on the OS kernel allowing this size and how many threads you have running.

23.6.2.3  x86/x64 Solaris

The default initial heap is mapped at address #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 #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.

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.

23.6.2.4  Windows and Macintosh

LispWorks (32-bit) for Windows and LispWorks (32-bit) for Macintosh both map by default at #x20000000. Since these platforms support reservation, normally you will not need to do anything special about this.

Problems may however arise if LispWorks operates in conjunction with non-relocatable software which insists on using addresses at #x20000000 or some distance above, in which case you will need to relocate LispWorks.
LispWorks (32-bit) for Windows can in principle grow up to some distance below \#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.

### 23.6.3 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).

#### 23.6.3.1 Linux

In LispWorks (64-bit) for Linux the range is 192GB, ending at \#x7000000000, because Linux cannot map above \#x8000000000 and puts 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.
23.6.3.2 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.

23.6.3.3 Windows and Macintosh

In LispWorks (64-bit) for Windows and LispWorks (64-bit) for Macintosh 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.

23.7 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 18, “Dynamic Data Exchange”.


23.8 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*.

### 23.9 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`.

### 23.10 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:

```lisp
(quit :status 42)
```

### 23.11 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 605. 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 Section 23.12 for information about universal binaries on Mac OS X.

### 23.12 Universal binaries on Mac OS X

The supplied 32-bit LispWorks for Macintosh images are universal binaries, which run the correct native architecture on PowerPC and Intel-based Macintosh computers by default.
A running Lisp image only supports one architecture, chosen when the image was started. On a PowerPC based Macintosh, this is always the PowerPC architecture. On an Intel-based Macintosh, it can be either the native Intel architecture or the PowerPC architecture (using Rosetta).

Functions such as `save-image` and `deliver` mentioned in Section 23.11 create an image containing only the running architecture and functions that operate on fasl files such as `compile-file` and `load` only support the running architecture.

To build a universal binary application from LispWorks for Macintosh 5.x, you will need to install LispWorks on an Intel-based Macintosh computer.

Building a new universal binary requires three steps:

1. Build the application for PowerPC.
   This can be done on your Intel machine using Rosetta
2. Build the application for Intel.
3. Combine the two applications to make a universal binary.

These steps can be automated on a single Intel-based Macintosh by creating a script that compiles and loads the application and then saves the image. Loading this by running LispWorks with the `-build` command line argument would save an image containing a single architecture, but you can use the same script to save a universal binary by calling `save-universal-from-script`, page 615.

**Note:** You may install LispWorks on multiple machines for use at the same time only if you own multiple LispWorks licenses.

### 23.13 User Preferences

LispWorks provides an API for setting and querying persistent per-user settings in a platform-dependent registry.

#### 23.13.1 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 314.)

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 `(setq product-registry-path)` and read it using `product-registry-path`.

### 23.13.2 Accessing persistent settings

Get and set preferences under the product path at runtime with `user-preference` and `(setq user-preference)`.

### 23.13.3 Example using user preferences

Define a registry path:

```
(setq (sys:product-registry-path :deep-thought)
       '("Software" "My Company" "Deep Thought"))
```

Store a preference for the current user:

```
(setq (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)
```

### 23.14 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:

```lisp
(defun collect-control-names (&key insertable
                             (max-name-size 256)
                             (max-names most-positive-fixnum))
  (win32:collect-registry-subkeys
   "CLSID"
   :root :root
   :max-name-size max-name-size
   :max-names max-names
   :value-function
   #'(lambda (hKeyClsid ClassidName)
       (win32:with-registry-key
         (hkeyX ClassidName :root hKeyClsid :errorp nil)
         (when (and
                 (win32:registry-key-exists-p "Control"
                  :root hkeyX)
                 (if insertable
                     (win32:registry-key-exists-p "Insertable"
                      :root hkeyX)
                     t)))
         (when-let
           (progid (win32:query-registry-value "ProgID" nil
                                    :root hkeyX
                                    :errorp nil))
           (values
            (list
             (win32:query-registry-value nil nil
                                          :root hkeyX)
             progid
             (win32:query-registry-value "InprocServer32" nil
                                      :root hkeyX
                                      :errorp nil))
           t)))
))
```

### 23.15 The home directory

This section describes the implementation of the Common Lisp function `user-homedir-pathname`. 
On Unix-based systems, the home directory is looked up using the C function getpwuid.

On Microsoft Windows systems, `user-homedirpathname` 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

```lisp
(setf environment-variable)
```

## 23.16 Special Folders

On Microsoft Windows and Mac OS X 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 Vista:

C:\ProgramData

Windows XP:

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

Windows 2000:

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

Mac OS X:

/Library/Application Support

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:

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

Here is another example of differences between operating systems. On Windows Vista:
On Windows 98 SE:

\[
\text{(sys:get-folder-path :my-documents)}\\
\rightarrow\\
\#P"C:/Users/dubya/Documents/"
\]

On Mac OS X:

\[
\text{(sys:get-folder-path :my-documents)}\\
\rightarrow\\
\#P"/u/ldisk/dubya/Documents/"
\]

See \textit{get-folder-path, page 1114} 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 \textit{get-user-profile-directory, page 1116}.  

\[
\text{(sys:get-folder-path :my-documents)}\\
\rightarrow\\
\#P"C:/Users/dubya/Documents/"
\]

\[
\text{(sys:get-folder-path :my-documents)}\\
\rightarrow\\
\#P"/u/ldisk/dubya/Documents/"
\]
LispWorks’ Operating Environment
This chapter summarises the technical differences between 64-bit LispWorks and 32-bit LispWorks. Both are ANSI Common Lisp implementations and support the language same extensions and libraries so in many ways they behave the same. However the programmer should be aware of the differences mentioned here.

24.1 Introduction

64-bit LispWorks has a larger address space, subject to physical memory. The maximum heap sizes are shown in Table 24.1.

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 24.2.

Other differences in 64-bit LispWorks are noted in the remaining sections of this chapter.
24.2 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 24.1.

Table 24.1 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>#x4000000000 to #x40000000000 (3.75TB)</td>
<td></td>
</tr>
<tr>
<td>Linux</td>
<td>#x40000000000 to #x7000000000 (192GB)</td>
<td>Effective limit around 160GB.</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>

64-bit LispWorks is relocatable on all supported platforms as described in “Startup relocation of 64-bit LispWorks” on page 310.

In contrast, 32-bit LispWorks has a maximum heap size of 1.5-3.0GB depending on platform and is relocatable on non-Windows platforms only, as described in “Startup relocation of 32-bit LispWorks” on page 307.

24.3 Architectural constants

Common Lisp constants have the values shown in Table 24.2

Table 24.2 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>
</tbody>
</table>
24.4 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.

24.5 Memory Management

Memory layout and the garbage collector (GC) differs significantly between the two implementations.

For the details see “Memory Management in 32-bit LispWorks” on page 104 and “Memory Management in 64-bit LispWorks” on page 112.

24.6 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.

24.7 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.

Table 24.2 Architectural constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>32-bit LispWorks</th>
<th>64-bit LispWorks</th>
</tr>
</thead>
<tbody>
<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.
Third party libraries loaded into 32-bit LispWorks must be 32-bit.
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 14, “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.
Description

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

Function

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 checks that initargs passed to it are valid.

Note: class-extra-initargs is useful only in complex cases. In most cases other ways of extending the set of valid initargs are simpler and clearer, such as the :extra-initargs class option, described in defclass.

Example In this session an illegal initarg :my-keyword is passed, causing make-instance to signal an error.

Then :my-keyword is added as an extra initarg, after which make-instance accepts it.
CL-USER 38 > (defclass my-class () ((a :initform nil)))
#<STANDARD-CLASS MY-CLASS 113AAA2F>

CL-USER 39 > (make-instance 'my-class :my-keyword 8)
Error: MAKE-INSTANCE is called with unknown keyword
:MY-KEYWORD among the arguments (MY-CLASS :MY-KEYWORD 
8) {no keywords allowed}
  1 (continue) Ignore the keyword :MY-KEYWORD
  2 (abort) Return to level 0.
  3 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed,
or :? for other options

CL-USER 40 : 1 > :a

CL-USER 41 > (defmethod clos:class-extra-initargs
   ((x my-class))
   '(:my-keyword))
#<STANDARD-METHOD CLOS:CLASS-EXTRA-INITARGS (MY-CLASS) 
1137C763>

CL-USER 42 > (make-instance 'my-class :my-keyword 8)
#<MY-CLASS 11368963>

See also
compute-class-potential-initargs
defclass
make-instance
set-make-instance-argument-checking

compute-class-potential-initargs

Summary
Computes the valid initargs of a class.

Package
clos

Signature
compute-class-potential-initargs class => initargs

Arguments
class A class.
Values

initargs A list of symbols, or t.

Description

The generic function `compute-class-potential-initargs` is called to compute the initialization arguments of a class. This set of valid initargs is used by `make-instance` when its arguments are checked.

class is the class passed to `make-instance`. That is, `compute-class-potential-initargs` specializes on the metaclass.

initargs is either a list of valid initargs, or t meaning that any initialization argument is allowed.

There is a supplied method on t, which returns nil.

The other supplied method is on `standard-class`. This consults the Relevant Methods, which are the applicable methods of `make-instance`, `allocate-instance`, `initialize-instance` and `shared-initialize`. If any of the Relevant Methods have a lambda list containing `&allow-other-keys` then initargs is t. Otherwise initargs is a list containing:

- all the &key arguments from Relevant Method lambda lists, and
- the initargs of the slots of class and its superclasses, and
- any extra initargs specified via the class option :extra-initargs (see `defclass` for details of this), and
- any extra initargs returned by `class-extra-initargs`.

The list initargs contains no duplicates, and the result of `compute-class-potential-initargs` is cached so that it is not recomputed unless one of the Relevant Methods, the class or its class precedence list is altered.

See also

- `class-extra-initargs`
- `make-instance`
- `set-make-instance-argument-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.

**funcallable-standard-object**  
*Class*

**Package**  
clos

**Superclasses**  
function
standard-object

**Subclasses**  
generic-function
The metaclass `funcallable-standard-object` provides the default `:direct-superclasses` for instances of `funcallable-standard-class` and its subclasses.

`funcallable-standard-object` is implemented as described in AMOP except for a different order in the class precedence list.

In AMOP the class precedence list is

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

whereas in LispWorks the class precedence list is

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

LispWorks is like this to be compliant with the rules in the ANSI Common Lisp Standard.

The AMOP class precedence list implies a class precedence for `generic-function` which violates the last sentence in ANSI Common Lisp 4.2.2 Type Relationships. See [www.lisp-works.com/reference/HyperSpec/Body/04_bb.htm](http://www.lisp-works.com/reference/HyperSpec/Body/04_bb.htm).

---

### process-a-class-option

**Generic Function**

**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:

(defun 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:

(defun 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:

(defun 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))

(defclass m2 (standard-class)
  ((titles-list :initarg :list-of-possible-titles)))
If the titles are to be evaluated, add a method like this:

```lisp
(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:

```lisp
(defmethod clos:process-a-class-option
  ((class m2)
   (name (eql :list-of-possible-titles))
   value)
  (list name ~`,value))
```

```lisp
(defclass my-multi-titled-class ()
  ()
  (:metaclass m2)
  (:list-of-possible-titles
   "Initial Title 1"
   "Initial Title 2")
)
```

See also

- `defclass`
- `process-a-slot-option`

---

**process-a-slot-option**

*Generic Function*

**Summary**: Describes how a `defclass` slot option is parsed.

**Package**: clos

**Signature**: `process-a-slot-option metaclass option value already-processed-other-options slot => processed-options`

**Arguments**:

- `metaclass` The metaclass of the class being parsed.
- `option` The slot option name.
- `value` The value of the slot option.
already-processed-other-options
A plist of initargs for non-standard options that have been processed already.

slot
The whole slot description.

Values
processed-options
A plist of initargs.

Description
The generic function `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))

(defun 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))
  (:metaclass extended-class))

To add a slot option :special-reader whose value is a non-evaluated symbol naming a reader:

(defun 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))
          already-processed-options)
        (list* :special-reader `(,value)
               already-processed-options)))
    (call-next-method)))

See also  defclass
          process-a-class-option

set-make-instance-initarg-checking

Function

Summary  Switches initarg checking in make-instance on or off.

Package  clos

Signature  set-make-instance-initarg-checking on => on

Arguments  on          A boolean.

Description  The function set-make-instance-initarg-checking provides control over whether make-instance checks its initialization arguments.

Calling set-make-instance-initarg-checking with on true, causes make-instance to check the initargs. This is the initial state of LispWorks.

Initarg checking is switched off globally and dynamically by (set-make-instance-initarg-checking nil).
Notes

The effect of calling `set-make-instance-initarg-checking` can be overridden in a runtime by the deliver argument `:make-instance-keyword-check`. See the *LispWorks Delivery User Guide* for details.

See also

- `class-extra-initargs`
- `compute-class-potential-initargs`
- `make-instance`

### slot-boundp-using-class

*Generic Function*

**Summary**

Implements `slot-boundp`.

**Package**

clos

**Signature**

```
slot-boundp-using-class class object slot-name => result
```

**Arguments**

- `class`  
  A class metaobject, the class of `object`.

- `object`  
  An object.

- `slot-name`  
  A slot name.

**Values**

- `result`  
  A boolean.

**Description**

The generic function `slot-boundp-using-class` implements the behavior of the `slot-boundp` function.

The implementation is as described in AMOP, except that the third argument is the slot name, and not a slot definition metaobject. The primary methods specialize on `t` for this argument.

See also

- `slot-makunbound-using-class`
- `slot-value-using-class`
**slot-makunbound-using-class**  
*Generic Function*

Summary  Implements `slot-makunbound`.

Package  `clos`

Signature  `slot-makunbound-using-class class object slot-name => object`

Arguments  
- `class`  A class metaobject, the class of `object`.
- `object`  An object.
- `slot-name`  A slot name.

Values  `object`  The `object` argument.

Description  
The generic function `slot-makunbound-using-class` implements the behavior of the `slot-makunbound` function. The implementation is as described in AMOP, except that the third argument is the slot name, and not a slot definition metaobject. The primary methods specialize on `t` for this argument.

See also  
- `slot-boundp-using-class`
- `slot-value-using-class`

**slot-value-using-class**  
*Generic Function*

Summary  Implements `slot-value`.

Package  `clos`

Signature  `slot-value-using-class class object slot-name => value`

Arguments  
- `class`  A class metaobject, the class of `object`.  

(setf `slot-value-using-class`) `value class object slot-name => value`
An object.

A slot name.

The value of the slot named by slot-name.

The generic function slot-value-using-class implements the behavior of the slot-value function.

The implementation is as described in AMOP, except that the third argument is the slot name, and not a slot definition metaobject. The primary methods specialize on t for this argument.

Note: by default, standard slot accessors are optimized to not call slot-value-using-class. This can be overridden with the :optimize-slot-access class option. See defclass for details.

See also:
- defclass
- slot-boundp-using-class
- slot-makunbound-using-class

### trace-new-instances-on-access

**Function**

Traces new instances of a given class, based on access modes.

**Summary**

**Package**

clos

**Signature**

trace-new-instances-on-access class-designator
&key read write slot-names break when process trace-output entrycond eval-before before backtrace => t

**Arguments**

- **class-designator** The class to trace.

**Values**

Returns t.
Description

Causes new instances of the class given by class-designator to be traced for the access modes given by read, write and slot-names.

The keyword arguments control which type of access are traced, and provide preconditions for tracing, code to run before access, and how to print any trace output. They are interpreted as described for trace-on-access.

This function, when used with the :break keyword, replaces the deprecated function break-new-instances-on-access.

Example

(trace-new-instances-on-access 'capi:display-pane :slot-names nil)

Suppose you have a bug whereby the slot bar of an instance of your class foo is incorrectly being set to a negative integer value. You could cause entry into the debugger at the point where the slot is set incorrectly by evaluating this form:

(clos:trace-new-instances-on-access 'foo :slot-names '(bar) :read nil :when '((and (integerp (car *traced-arglist*)) (< (car *traced-arglist*) 0))) :break t)

and running your program.

See also

break-new-instances-on-access
untrace-new-instances-on-access
trace-on-access

trace-on-access

Function

Summary

Invokes the trace facilities when an instance of a class is accessed.

Package

clos
Signature: \texttt{trace-on-access instance \&key read write slot-names break when process trace-output entrycond eval-before before backtrace => t}

Arguments:
- \texttt{instance}: A CLOS instance.
- \texttt{read}: A generalized boolean.
- \texttt{write}: A generalized boolean.
- \texttt{slot-names}: A list of symbols, or \texttt{t}.
- \texttt{break}: A generalized boolean.
- \texttt{when}: A form.
- \texttt{process}: A form.
- \texttt{trace-output}: A form.
- \texttt{entrycond}: A form.
- \texttt{eval-before}: A list of forms.
- \texttt{before}: A list of forms.
- \texttt{backtrace}: A keyword, \texttt{t} or \texttt{nil}.

Values: Returns \texttt{t}.

Description: A useful debugging function which causes access to \texttt{instance} to invoke the trace facilities. Accesses include calls to \texttt{slot-value} and accessor functions defined by the class of \texttt{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 \texttt{trace} macro (but note that these CLOS symbols are functions, so the keyword values are evaluated immediately, unlike in \texttt{trace}).

\texttt{read} controls whether reading slots is traced. The default is \texttt{t}.

\texttt{write} controls whether writing slots is traced. The default is \texttt{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

Function

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

Function

Summary
Removes the trapping installed by break-on-access. Note that this function is deprecated.

Package
clos

Signature
unbreak-on-access instance

Arguments
instance A class instance

Values
Returns t.

Description
Removes any break installed on instance by break-on-access. See untrace-on-access for details.

Note: this function is deprecated. You should now use untrace-on-access instead.

See also
untrace-on-access

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
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. The TCP/IP mechanism allows LispWorks to connect to or implement a server. It also allows using Secure Sockets Layer (SSL) processing in the socket.

Before the interface can be used the module "comm" must be loaded using

```
(require "comm")
```

---

**attach-ssl**

*Function*

**Summary**

Attaches SSL to a socket stream.

**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.
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

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

### `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`.

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.

**Signature**

`detach-ssl socket-stream &key retry-count retry-timeout`

**Arguments**

- `socket-stream` A `socket-stream`.
- `retry-count` A non-negative integer.
- `retry-timeout` A non-negative real.

**Description**

The function `detach-ssl` detaches the SSL from the socket-stream `socket-stream`. If `socket-stream` is not attached to an SSL, `detach-ssl` just returns immediately. Otherwise, it detaches the SSL from `socket-stream`, tries to shut down the SSL cleanly, and then frees the objects that were allocated by `attach-ssl`. `retry-count` specifies how many additional times to call `SSL_shutdown` after the second to attempt to get a successful shutdown. The default value of `retry-count` is 5.

`retry-timeout` specifies the time in seconds to wait between each of the calls to `SSL_shutdown`. If it fails to get a successful shutdown after these attempts, `detach-ssl` signals an error. The default value of `retry-timeout` is 0.1.

Note that the shutdown calls happen after the SSL has been detached from `socket-stream` as far as LispWorks is concerned, so if an error occurs at this point and is aborted, `socket-stream` can be used in `attach-ssl` again (assuming that the peer can cope with this situation).
If `retry-count` is `nil`, `detach-ssl` does not try to get a successful shutdown call. This value is used when the stream is closed, but should not be used normally.

See also `attach-ssl`

### do-rand-seed

**Function**

**Summary**
Calls the SSL function `RAND_seed`.

**Package**
`comm`

**Signature**
do-rand-seed

**Arguments**
do-rand-seed takes no arguments.

**Values**
do-rand-seed returns no values.

**Description**
The function `do-rand-seed` calls the SSL function `RAND_seed` with some suitable value, which is dependent in a non-trivial way on the current time, the history of the current process and the history of the machine it is running on.

If the machine that it runs on has the file `/dev/urandom`, do-rand-seed does nothing.

See also `attach-ssl`

### ensure-ssl

**Function**

**Summary**
Initializes SSL.

**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 287. This default may be changed by calling set-ssl-library-path.

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

get-host-entry    Function

Summary    Returns address or name information about a given host.

Package    comm

Signature    get-host-entry host &key fields => field-values

Arguments    host    A number or a string.
              fields    A list of keywords.

Values    field-values    Values, one for each field.

Description    Using whatever host naming services are configured on the current machine, get-host-entry returns address or name
information about the given host. \texttt{nil} is returned if the host is unknown.

The \texttt{host} argument can be one of the following:

- a name string, for example "\texttt{www.foobar.com}"
- a dotted inet address string, for example 
  \texttt{"209.130.14.246"}
- a integer representing the inet address, for example 
  \texttt{#xD1820EF6}

The \texttt{fields} argument is a list of keywords describing what information to return for the host. If \texttt{get-host-entry} succeeds, it returns multiple values, one value for each field specified. The following fields are allowed:

\begin{itemize}
\item \texttt{:address} The primary IP address as an integer.
\item \texttt{:addresses} A list of all the IP addresses as integers.
\item \texttt{:name} The primary name as a string.
\item \texttt{:aliases} The alias names as a list of strings.
\end{itemize}

\textbf{Note:} although the results of \texttt{get-host-entry} are not cached by LispWorks, the operating System might cache them.

\textbf{Examples}

\begin{verbatim}
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")
\end{verbatim}
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 Connected sockets have two addresses, local and remote. The get-socket-address function returns the local address.

See also get-socket-peer-address
socket-stream-address

get-socket-peer-address

Function

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

Package comm

Signature get-socket-peer-address socket => address, port

Arguments socket A socket handle.

Values address The remote host address of the socket or nil if not connected.
**port**

The remote port number of the socket or `nil` if not connected.

**Description**

Connected sockets have two addresses, local and remote. The `get-socket-peer-address` function returns the remote address.

**See also**

`get-socket-address`

`socket-stream-peer-address`

---

**get-verification-mode**

**Function**

**Summary**

Returns the mode of the SSL.

**Package**

`com`

**Signature**

`get-verification-mode ssl-or-ssl-ctx => result`

**Arguments**

`ssl-or-ssl-ctx` A foreign pointer of type `ssl-pointer` or `ssl-ctx-pointer`.

**Values**

`result` A list of symbols.

**Description**

The function `get-verification-mode` returns the mode of the `ssl-pointer` or `ssl-ctx-pointer` as a list of symbols.

`result` is a list containing zero or more of the symbols:`:verify-client-once`,`:verify-peer` and`:fail-if-no-peer-cert`, corresponding to the C constants `VERIFY_CLIENT_ONCE` `VERIFY_PEER` and `FAIL_IF_NO_PEER_CERT` respectively.

**See also**

`set-verification-mode`
**ip-address-string**

*Function*

**Summary**
Returns the dotted IP address string from the integer IP address.

**Package**
`comm`

**Signature**
`ip-address-string ip-address => string-ip-address`

**Arguments**
`ip-address` An integer.

**Values**
`string-ip-address` The dotted string format of the given IP address.

**Description**
The `ip-address-string` function converts its argument to a string in the standard dotted IP address notation `a.b.c.d`.

**See also**
`string-ip-address`

---

**make-ssl-ctx**

*Function*

**Summary**
Makes a `SSL_CTX` object.

**Package**
`comm`

**Signature**
`make-ssl-ctx &key ssl-ctx ssl-side => ssl-ctx-ptr`

**Arguments**
`ssl-ctx` A symbol or a foreign pointer.

`ssl-side` One of the keywords `:client`, `:server` or `:both`.

**Values**
`ssl-ctx-ptr` A foreign pointer of type `ssl-ctx-pointer`.

**Description**
The function `make-ssl-ctx` first calls `ensure-ssl`, and returns a foreign pointer of type `ssl-ctx-pointer`. 
If the value of ssl-ctx is \texttt{t}, \texttt{:default}, \texttt{:v2}, \texttt{:v3}, \texttt{:v23} or \texttt{:tls-v1}, \texttt{make-ssl-ctx} creates a \texttt{SSL_CTX} object and returns a pointer to it.

The value of ssl-ctx can also be a foreign pointer of type \texttt{ssl-ctxt-pointer}, in which case it is simply returned. If \texttt{ssl-ctx} is a foreign pointer of type \texttt{ssl-pointer}, then \texttt{make-ssl-ctx} signals an error.

The meaning of the keyword arguments \texttt{ssl-ctx} and \texttt{ssl-side} is as described for \texttt{socket-stream}. The default value of \texttt{ssl-ctx} is \texttt{t} and the default value of \texttt{ssl-side} is \texttt{:server}.

See also

\begin{itemize}
  \item \texttt{ensure-ssl}
  \item \texttt{socket-stream}
  \item \texttt{ssl-ctxt-pointer}
\end{itemize}

\textbf{open-tcp-stream}\hfill\textit{Function}

\textbf{Summary} Attempts to connect to a socket on another machine and returns a stream object for the connection.

\textbf{Package} \texttt{comm}

\textbf{Signature} \\
\texttt{open-tcp-stream hostname service &key direction element-type errorp read-timeout write-timeout timeout ssl-ctx ctx-configure-callback ssl-configure-callback local-address local-port nodelay keepalive => stream-object}

\textbf{Arguments} \\
\begin{itemize}
  \item \texttt{hostname} An integer or string.
  \item \texttt{service} A string or a fixnum.
  \item \texttt{direction} One of :\texttt{input}, :\texttt{output} or :\texttt{io}.
  \item \texttt{element-type} \texttt{base-char} or a subtype of \texttt{integer}.
  \item \texttt{errorp} A boolean.
  \item \texttt{read-timeout} A positive number, or \texttt{nil}.
\end{itemize}
The open-tcp-stream function attempts to connect to a socket on another machine and returns stream-object for the connection if successful. The server machine to connect to is given by hostname, which can be one of the following:

- A string naming the host, for example "www.nowhere.com"
- A string providing the IP address, for example "204.71.177.75"
- An integer IP address in network order, for example #xCC47B14B

The name of the service to provide is given by service. If service is a string, the location of the file specifying the names of the services available varies, but typically on Windows 98 it
is called SERVICES and is stored in the Windows directory, and on Windows NT-based systems it is the file

%SystemRoot%\system32\drivers\etc\SERVICES

The service can also be a fixnum representing the port number of the desired connection.

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 string or integer value is interpreted as for hostname 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`.

**Example**

The following example opens an HTTP connection to a given host, and retrieves the root page:

```
(with-open-stream (http (comm:open-tcp-stream
   "www.lispworks.com" 80))
   (format http "GET / HTTP/1.0"
      (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**

- `start-up-server`
- `socket-stream`

### openssl-version

**Function**

**Summary**

Returns the version of the loaded OpenSSL library.

**Package**

`comm`
Signature  \texttt{openssl-version} \texttt{\&optional what => result}

Arguments \texttt{what} One of the keywords \texttt{version}, \texttt{directory}, \texttt{platform}, \texttt{cflags} and \texttt{built-on}.

Values \texttt{result} A string.

Description The function \texttt{openssl-version} returns a string specifying the version of the loaded OpenSSL library.

The argument \texttt{what} takes these values:

\begin{itemize}
  \item \texttt{version} \texttt{result} is the version string, which looks like:
    \begin{itemize}
      \item "OpenSSL 0.9.7i 14 Oct 2005"
      \item "OpenSSL 0.9.8a 11 Oct 2005"
    \end{itemize}
  \item \texttt{built-on} Returns a string specifying when it was built.
  \item \texttt{directory} Returns where OpenSSL thinks it is installed.
  \item \texttt{platform} Returns OpenSSL’s idea of which platforms it is.
  \item \texttt{cflags} The compilation command.
\end{itemize}

The default value of \texttt{what} is \texttt{version}.

See also \texttt{ensure-ssl}

\textbf{pem-read} \textit{Function}

Summary An interface to the SSL \texttt{PEM_read_bio_*} functions.

Package \texttt{comm}

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

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

Values

result             A foreign pointer or nil.

Description

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

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

filename specifies the file to load.

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

If callback is non-nil, it must be a function with signature:

callback maximum-length rwflag => pass-phrase

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

If it succeeds, pem-read returns a foreign pointer to the structure that was returned by the pem function. If pem-read fails, if errorp is non-nil it signals an error, otherwise it returns nil. The default value of errorp is nil.

read-dhparams

Function

Summary

Reads or uses cached SSL DH parameters.
### Description

The function `read-dhparams` reads or uses cached DH parameters.

`filename` specifies the file to check.

Unless `force` is true, `read-dhparams` checks if 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`
### set-verification-mode

**Function**

**Summary**
Sets the verification mode for CTX.

**Package**
comm

**Signature**
```
set-verification-mode ssl-ctx ssl-side mode &optional callback
```

**Arguments**
- `ssl-ctx`: A foreign pointer of type `ssl-pointer` or `ssl-ctx-pointer`.
- `ssl-side`: `:server` or `:client`.
- `mode`: An integer, one of the symbols `:never`, `:always`, `:once`, or a list of keywords.
- `callback`: A foreign function.

**Values**
- `result`: A list of symbols.

**Description**
The function `set-verification-mode` sets the verification mode for CTX according to arguments `ssl-side` and `mode`.

When `ssl-side` is `:server`, `mode` can be:

- **An integer**
  - `mode` is passed directly to `SSL_set_verify` or `SSL_CTX_set_verify`.

- **`: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

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 (corre-
spending 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 desig-
nator 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 signa-
ture:
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

Summary
Sets the options in a SSL_CTX.

Package
comm

Signature
set-ssl-ctx-options ssl-ctx &key microsoft_sess_id_bug
netscape_challenge_bug netscape_reuse_cipher_change_bug
ssleay2_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_fragments 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:  

`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**

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 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**

*Class*

**Summary**

The condition class for socket errors.
Superclasses  simple-error
Subclasses  ssl-condition
Initargs  :stream  A socket-stream.
Description  The condition class for socket errors.

socket-stream  

Class

Summary  The socket stream class.

Superclasses  buffered-stream

Initargs  :socket  A socket handle.
:direction  One of :input, :output, or :io.
:element-type  An element type.
: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
The `socket-stream` class 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` keyword 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 `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 `socket-stream`: the character data is constructed from the input as if by `code-char`. Similarly, `write-sequence` can be called with a string buffer and a binary `socket-stream`: the output is converted from the character data as if by `char-code`. Also, 8-bit binary data can be read and written to a `base-char socket-stream`.

All standard stream I/O functions except for `write-byte` and `read-byte` have this flexibility.

The `:read-timeout` initarg specifies the read timeout in seconds, or is `nil`, meaning there are no timeouts during reads (this is the default).

The `read-timeout` property is intended for use when a socket connection might hang during a call to any Common Lisp input function. The `read-timeout` can be set by `make-instance` or by `open-tcp-stream`. It can also be modified by `(setf stream:stream-read-timeout)`. When `read-timeout` is `nil`, there is no timeout during reads and the call may hang. When `read-timeout` is not `nil`, and there is no input from the socket for more than `read-timeout` seconds, any reading function returns `end-of-file`. The `read-timeout` does not limit the time inside `read`, but the time between successful extractions of data from the socket. Therefore, if the reading needs several rounds it may take longer than `read-timeout`. 
Using `(setf stream:stream-read-timeout)` on the stream while it is inside a read function has undefined effects. However, the setf function can be used between calls to read functions. The `read-timeout` property of a stream can be read by

```
(stream:stream-read-timeout stream)
```

The `:write-timeout` initarg specifies the write timeout in seconds, or is `nil`, meaning that there are no timeouts during writes (this is the default).

The `write-timeout` property is similar to `read-timeout`, but for write operations. If flushing the stream buffer takes too long then `error` is called.

The keyword arguments `:ssl-ctx`, `:ssl-side`, `:ctx-configure-callback` and `:ssl-configure-callback` can be be passed to create and configure socket streams with SSL processing.

`ssl-ctx`, if non-nil, specifies that the stream uses SSL and further specifies the `SSL_CTX` object to use. The value of `ssl-ctx` can be a symbol which, together with `ssl-side`, specifies which protocol to use. The value `t` or `:default` means use the default, which is currently the same as `:v23`. The values `:v2`, `:v3`, `:v23` and `:tls-v1` are mapped to the `SSLv2_*`, `SSLv3_*`, `SSLv23_*` and `TLSv1_*` methods respectively. With these symbol values of `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 speci-
fies the SSL to use. This maybe a result of a call to \texttt{ssl-new} but can also be the result of user code, provided that it points to a valid SSL object and has the type \texttt{ssl-pointer}. The SSL is used and is not freed when the stream is closed.

When you pass a \texttt{ssl-ctx-pointer} or a \texttt{ssl-pointer} foreign pointer as the \texttt{ssl-ctx} argument, it must have already been set up correctly.

\textit{ssl-side} specifies which side the socket stream is. The value of \textit{ssl-side} is used in two cases:

- When a new \texttt{SSL_CTX} object is created, it is used to select the method:
  \begin{verbatim}
  :client => *_client_method
  :server => *_server_method
  :both   => *_method
  \end{verbatim}

- When a new SSL object is created, when \textit{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 \textit{ssl-ctx} is a \texttt{ssl-pointer}, \textit{ssl-side} is ignored.

\textit{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 \texttt{SSL_CTX} is created. If the value of \texttt{ssl-ctx} is not a symbol, \textit{ctx-configure-callback} is ignored.

\textit{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}, \textit{ssl-configure-callback} is ignored.

\textbf{Example} The following makes a bidirectional stream connected to a socket specified by \textit{handle}.

\begin{verbatim}
(make-instance 'comm:socket-stream
  :socket handle
  :direction :io
  :element-type 'base-char)
\end{verbatim}
This example creates a socket stream with a read-timeout:

\[
\text{(make-instance 'comm:socket-stream}
  \text{ :handle handle}
  \text{ :direction :input}
  \text{ :read-timeout 42)}
\]

The following form illustrates character I/O in a binary socket-stream:

\[
\text{(with-open-stream (x}
  \text{ (comm:open-tcp-stream}
    \text{ "localhost" 80}
    \text{ :element-type '(unsigned-byte 8)})
  \text{ (write-sequence (format nil "GET / HTTP/1.0-%\n") x)}
  \text{ (force-output x)}
  \text{ (let ((res (make-array 20 :element-type 'base-char))}
               \text{ (values (read-sequence res x) res))})
\]

The following form illustrates binary I/O in a base-char socket-stream:

\[
\text{(with-open-stream (x}
  \text{ (comm:open-tcp-stream}
    \text{ "localhost" 80}
    \text{ :element-type 'base-char})
  \text{ (write-sequence}
    \text{ (map '(simple-array (unsigned-byte 8) 1)
       'char-code}
    \text{ (format nil "GET / HTTP/1.0-%\n")})
    \text{ x})
  \text{ (force-output x)}
  \text{ (let ((res (make-array 20 :element-type}
               \text{ '(unsigned-byte 8)))}}
  \text{ (values (read-sequence res x)}
  \text{ (map 'string 'code-char res))})
\]

See also
open-tcp-stream
start-up-server
stream-read-timeout
wait-for-input-streams
socket-stream-address  Function

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

Package  comm

Signature  

Arguments  stream  A socket stream.

Values  

Description  Connected socket streams have two addresses, local and remote. The socket-stream-address function returns the local address.

See also  

socket-stream-ctx  Function

Summary  Accesses the SSL_CTX attached to a socket stream.

Package  comm

Signature  

Arguments  socket-stream  A socket-stream.

Values  ssl-ctx-pointer  A foreign pointer of type ssl-ctx-pointer, or nil.
The function `socket-stream-ctx` accesses the SSL_CTX that is attached to the socket-stream socket-stream.

It returns `nil` if SSL is not attached.

See also `socket-stream`
`ssl-ctx-pointer`

### `socket-stream-peer-address`

**Function**

**Description**

Connected socket streams have two addresses, local and remote. The `socket-stream-peer-address` function returns the remote address.

**See also**

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

**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**

The function `socket-stream-peer-address` accesses the SSL attached to a socket stream.

**See also**

`socket-stream-ssl`

**Summary**

Accesses the SSL attached to a socket stream.
**socket-stream-ssl**

**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`.

**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.

**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`.

**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`.

**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 an 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  \textit{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  \textit{Class}

Summary

The class for SSL errors corresponding to SSL_ERROR_WANT_X509_LOOKUP.

Superclasses

ssl-condition

Description

The condition class ssl-x509-lookup corresponds to SSL_ERROR_WANT_X509_LOOKUP. It happens when a certificate is rejected by a user callback.
**start-up-server**

*Function*

**Summary**
Starts a TCP server.

**Package**
`comm`

**Signature**
```
start-up-server &key function announce service address 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`.
- `address` An integer, a string or `nil`.
- `nodelay` A generalized boolean.
- `keepalive` A generalized boolean.
- `process-name` A symbol or expression.
- `wait` A boolean.
- `error` A boolean.

**Values**
- `process` A process, or `nil`.
- `startup-condition` A condition object, or `nil`.

**Description**
The `start-up-server` function starts a TCP server. Use `process-kill` to kill the server, and `open-tcp-stream` to send messages from another client to the server.

The `function` argument provides the name of the function that processes connections. When a connection is made `function` is called with the connected socket handle, at which point you can make a stream using `make-instance` and communicate with the client. The server does not accept more connections until `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 `function` is not specified the built-in Lisp listener server is used. See the examples section below.

If `announce` is a stream or `t` (denoting `*standard-output*`), a message appears on the stream when the server is started.

If `announce` is a function it is called when the server is started. `announce` should take two arguments: `socket` and `condition`. `socket` is the socket used by the server: `announce` can therefore be used to record this socket. `condition` describes the error if there is one. `announce` can be called with `socket nil` and a condition only if `error` is `nil`. If the process is killed, `announce` is called with `socket nil` and `condition nil`.

The default for `announce` is `nil`, meaning there is no message.

If `service` is a string or positive integer, it specifies the name of the service. The location of the file specifying the names of services available varies, but typically on Windows 98 it is called `SERVICES` and is stored in the `Windows` directory, and on Windows NT-based systems it is the file

```
%SystemRoot%\system32\drivers\etc\SERVICES
```

If `service` is `nil` or 0, then `start-up-server` chooses a free port. The default value for `service` is "lispworks".

If `address` is a string or integer that can be resolved to an IP address, then the server only receives connections for that IP address. This must be one of the addresses associated with the machine and allowed values are a string naming a host, such as `"www.nowhere.com"`, a string providing the IP address, such as `"204.71.177.75"`, or and integer IP address in network order, such as `#xCC47B14B`.

If `address` is `nil` or 0, then the server will receive connections to all IP addresses on the machine. This is the default.

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 signalled, `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`.

The `error` argument controls what happens if an error is signalled 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)`.

**Note:** 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.

**Examples**

The following example uses the built-in Lisp listener server:

```
(comm:start-up-server :service 10243)
```

It makes a Lisp listener server on port 10243 (check with local network managers that this port number is safe to use). When a client connects to this, Lisp calls `read`. The client should send a string using Common Lisp syntax followed by a newline. This string is used to name a new light-weight process that runs a Lisp listener. When this has been created, the server waits for more connections.

The next example illustrates the use of the `function` argument. For each line of input read by the server it writes the line back with a message. The stream generates `EOF` if the other end closes the connection.

```
(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'-%" 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
socket-stream

start-up-server-and-mp

Function

Package    comm

Signature  start-up-server-and-mp &key function announce service address process-name

Arguments  function       A function name.
announce     An output stream, t, nil or a function.
The start-up-server-and-mp function starts multiprocessing if it has not already been started and then calls start-up-server with the supplied function, announce, service, address and process-name arguments.

Note: 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 the integer IP address from the given dotted IP address string.

Package comm

Signature \texttt{string-ip-address ip-address-string} => \texttt{ip-address}

Arguments \texttt{string-ip-address} A string denoting an IP address in dotted format.

Values \texttt{ip-address} An integer IP address.

Description The \texttt{string-ip-address} function takes a string in the standard dotted IP address notation \texttt{a.b.c.d} and returns the corresponding integer IP address.

See also \texttt{ip-address-string}
with-noticed-socket-stream

Macro

Package comm

Signature with-noticed-socket-stream (stream) &body body

Arguments

- stream: A stream created using open-tcp-stream.
- body: Code to be executed while the stream is “noticed”.

Description

The macro with-noticed-socket-stream evaluates the body forms with the stream stream "noticed" for input. stream becomes unnoticed afterwards.

The macro is designed to be used with streams created by open-tcp-stream.

Notes

1. You do not normally need to use this macro, because all of the standard functions that read from socket streams (read-char and so on) will do this automatically when necessary. However, if you call process-wait yourself with a wait-function that detects new input from a socket stream, then this macro is necessary to cause LispWorks to evaluate the wait-function when there is input on the underlying socket. Without that, there might be a delay before the thread responds to the input.

2. with-noticed-socket-stream is not implemented on the Windows platform.

See also open-tcp-stream
The COMM Package
The COMMON-LISP Package

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 (an HTML version, the Common Lisp Hyperspec, is supplied with LispWorks) for full documentation about standard Common Lisp symbols.

apropos Function

Summary Searches for interned symbols.

Package common-lisp

Signature apropos string &optional package external-only => <no values>

Arguments string A string designator.
  package A package designator or nil.
  external-only A generalised boolean.
The function `apropos` behaves as specified in ANSI Common Lisp. There is an additional optional argument `external-only`, which if true restricts the search to symbols which are external in the searched package or packages. The default value of `external-only` is `nil`.

See also `apropos-list`

*describe-print-length*  
*describe-print-level*  
`regexp-find-symbols`

---

**apropos-list**

**Function**

**Summary**

Searches for interned symbols.

**Package**

`common-lisp`

**Signature**

`
apropos-list string &optional package external-only => symbols`

**Arguments**

`string` A string designator.

`package` A package designator or `nil`.

`external-only` A generalised boolean.

**Values**

`symbols` A list of symbols.

**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**

*Type*

**Summary**
The base string type.

**Package**
common-lisp

**Signature**
base-string length

**Arguments**
length The length of the string (or *, meaning any).

**Description**
The type of base strings.

---

**close**

*Generic Function*

**Summary**
The close function is implemented as a generic function.

**Package**
common-lisp

**Signature**
close stream &key abort => result

**Arguments**
stream A stream.
abort A generalized boolean.

**Values**
result A boolean.

**Description**
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`, 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 pri-
mary method specialized on a subclass of `buffered-stream`
is that it must close any underlying data source and return
true.

The `close` method on the `fundamental-stream` class sets a
flag for `open-stream-p`

See also `buffered-stream`
`fundamental-stream`
`open-stream-p`

---

**coerce**

Summary

Extends the standard `coerce` function, allowing it to take any
Common Lisp type specifier.

Package `common-lisp`

Signature `coerce object result-type => result`

Arguments

- `object` A Lisp object.
- `result-type` A type specifier.

Values

- `result` An object of type `result-type`

Description

The `coerce` function still performs those conversions
required by the standard, but a larger set of type specifiers is
allowed for coercion. A `type-error` is signalled if `result` can-
not be returned as the `result-type` specifies.

See also `concatenate`
compiles a lambda expression into a code vector.

Package common-lisp

Signature compile name &optional definition => name, function

Arguments definition If supplied, this is a lambda-expression to be compiled.
If not supplied, then the lambda-expression used is the current definition of the name (in this case name must be a non-nil symbol with an uncompiled definition).

name If not nil, this is the symbol that is to receive the compiled function as its global function definition.

Values A single value is returned, being the name symbol if supplied, or when name is nil the compiled function definition itself. Such compiled-function objects are not printable (but see disassemble) other than as #<compiled function for SYMBOL>.

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*. 
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**

Summary

Compiles a Lisp source file into a form that both loads and runs faster.

Package

common-lisp

Signature

```lisp
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.
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). 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. This argument is useful if you are using a non-default file extension for binary files. If you use a non-default file extensions for binary files, you must inform LispWorks of this by pushing the file extension string onto the variable sys::*binary-file-types*. If you fail to do this, LispWorks assumes that these files are text rather than compiled files. 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 \texttt{verbose}, messages are also printed about:

- compiler optimization settings before the file is processed, and
- multiple matches when \texttt{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 \texttt{*compile-verbose*}, which defaults to \texttt{t}.

\textit{print} controls the printing of information about the compilation. It can have the following values. If \textit{print} is \texttt{nil}, no information is printed. If \textit{print} is a non-positive number, then only warnings are printed. If \textit{print} is a positive number no greater than 1, or if \textit{print} is any non-number object, then the information printed consists of all warning messages and one line of information per function that is compiled. If \textit{print} is a number greater than 1, then full information is printed. The default value of \textit{print} is the value of \texttt{*compile-print*}, which has the default value 1.

Warning messages are printed to \texttt{*error-output*}. Other messages are printed to \texttt{*standard-output*}.

\textit{external-format} is interpreted as for \texttt{open}. The default value is \texttt{:default}.

If \textit{load} is true, then the file is loaded after compilation.

\textit{output-truename} is the true name of the output file, or \texttt{nil} if that cannot be created.

\textit{warnings-p} is \texttt{nil} if no conditions of type \texttt{error} or \texttt{warning} were detected during compilation. Otherwise \textit{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 note
In LispWorks 5.1 and previous versions, warning messages are printed to *standard-output*.

Examples

```lisp
(defun compile-file (filename) ; compile-filename to compiled-filename
  (let ((output-file (getvar 'output-file))) ; set output-file
    (format nil ;; compile-filename to compiled-filename
      (concatenate 'string output-file filename)))

(defun compile-file (filename) ; compile-filename to compiled-filename
  (let ((output-file (getvar 'output-file))) ; set output-file
    (format nil ;; compile-filename to compiled-filename
      (concatenate 'string output-file filename))))
```

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 27.1 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>.cfas1</td>
</tr>
<tr>
<td>x64 Windows/64-bit LispWorks</td>
<td>.64cfas1</td>
</tr>
<tr>
<td>x86 Linux/32-bit LispWorks</td>
<td>.ufas1</td>
</tr>
<tr>
<td>amd64 Linux/64-bit LispWorks</td>
<td>.64ufas1</td>
</tr>
<tr>
<td>x86 FreeBSD/32-bit LispWorks</td>
<td>.ffas1</td>
</tr>
<tr>
<td>HP-PA/32-bit LispWorks</td>
<td>.pfas1</td>
</tr>
<tr>
<td>SPARC/32-bit LispWorks</td>
<td>.wfas1</td>
</tr>
<tr>
<td>SPARC/64-bit LispWorks</td>
<td>.64wfas1</td>
</tr>
<tr>
<td>x86 Solaris/32-bit LispWorks</td>
<td>.sfas1</td>
</tr>
<tr>
<td>amd64 Solaris/64-bit LispWorks</td>
<td>.64sfas1</td>
</tr>
<tr>
<td>Intel Macintosh/32-bit LispWorks</td>
<td>.xfas1</td>
</tr>
<tr>
<td>PowerPC Macintosh/32-bit LispWorks</td>
<td>.nfas1</td>
</tr>
<tr>
<td>Intel Macintosh/64-bit LispWorks</td>
<td>.64xfas1</td>
</tr>
<tr>
<td>PowerPC Macintosh/64-bit LispWorks</td>
<td>.64nfas1</td>
</tr>
</tbody>
</table>

You can find the fasl file extension appropriate for your machine by looking at the variable `system:*binary-file-type*`. The variable `system::*binary-file-types*` contains a list of all the file extensions currently recognized by `load` and `load-data-file`.

Compatibility Note: In LispWorks for Windows 4.4 and previous, the fasl file extension is `.fsl`. This changed in LispWorks 5.0.
In LispWorks for Linux 4.4 and previous, 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 `concatenate` function 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 signalled.

See also

`coerce`

**declaim**  
*Macro*

**Summary**  
Established a specified declarations.

**Package**  
`common-lisp`
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`
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 following are extensions to the Common Lisp definition of declare:

- **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
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 optimization can be switched off using the :optimize-slot-access nil class option.
To add valid initialization arguments for the class, use the class option `:extra-initargs`. The argument passed via this option is evaluated, and should return a list of extra initialization arguments for the class. `make-instance` will treat these as valid when checking its arguments.

### Compatibility Note

When a class is redefined, its extra initargs are always reset.

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)))

(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:
(defclass a () ()
  (:extra-initargs '(:a-initarg)))
#<STANDARD-CLASS A 21C2E4FC>

(defclass b (a) () (:extra-initargs '(:b-initarg)))
#<STANDARD-CLASS B 2068573C>

(defclass c (a) ())
#<STANDARD-CLASS C 22829D44>

(make-instance 'b :a-initarg "A" :b-initarg "B")
#<B 2068BCE4>

(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
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 `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

```lisp
(defunpackage "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.
**test**  
Filtering test (only pathnames matching the test are collected).

**directories**  
A boolean controlling whether non-matching directories are included in the result.

**flat-file-namestring**  
A generalized boolean.

**link-transparency**  
If `nil`, then symbolic links are not followed. This means that returned names are not necessarily truenames, but has the useful feature that the `pathname-directory` of each pathname returned is the directory supplied as argument.

The default value of `link-transparency` is given by the special variable, `*directory-link-transparency*`, which has initial value `t` on UNIX/Linux/Mac OS X. By setting this variable to `nil`, you can get the old behavior of `directory`. On Windows, where the file system does not normally support symbolic links, this variable is initially `nil`.

**non-existent-link-destinations**  
If this is non-nil, then the pathname pointed to by a symbolic link appears in the output whether or not this file actually exists. If :link-transparency is non-nil and :non-existent-link-destinations is `nil` (this is the default on UNIX/Linux/Mac OS X), then symbolic links to nonexistent files do not appear.

The default value is `nil`.

**Values**  
**pathnames**  
A list of physical pathnames.
**Description**

*directory* collects all the pathnames matching the given pathname.

*directory* returns truenames, conforming to the ANSI specification for Common Lisp. Some programs may depend on the old behavior, however (and *directory* is slower if it has to find the truename for every file in the directory), and so two keyword arguments are available so that the old behavior can still be used: *link-transparency* and *non-existent-link-destinations*.

Because truenames are now returned, the entries . and .. no longer show up in the output of *directory*. This means, for instance, that

```
(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 sub-directories 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`.

**Note:** 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.

**Note:** 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`.

**Compatibility Note**

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 truenames.

**Example**

```lisp
CL-USER 16 > (pprint (directory "*.**"))

(#P"C:/Program Files/LispWorks/readme-4450.txt"
 #P"C:/Program Files/LispWorks/Msvcrt.dll"
 #P"C:/Program Files/LispWorks/LW4450.isu"
 #P"C:/Program Files/LispWorks/lispworks-4450.exe"
 #P"C:/Program Files/LispWorks/license-4450.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 pathname in its result when given a full-specified pathname:

CL-USER 1 > (directory
    (make-pathname :host "H"
        :device :unspecific
        :directory (list :absolute "tmp")
        :name :unspecific
        :type :unspecific
        :version :unspecific))

(#P"H:/tmp/"

The next two examples illustrate the effect of flat-file-namestring. Suppose the directory dir contains files interp.exe and file.lisp.

This call matches interp.exe, where the name interp ends with p, but does not match file.lisp, where the name file ends with e:

(directory "dir/*p")

The next call matches file.lisp, where the namestring file.lisp ends with p, but does not match interp.exe, where the namestring interp.exe ends with e:

(directory "dir/*p" :flat-file-namestring t)

See also 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*.

On UNIX and Mac OS X, the number of instructions in the disassembly is also printed, at the end.

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 signalled 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 and symbol 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.

**Package**

`common-lisp`

**Signature**

`documentation x doc-type => documentation`

`(setf documentation) new-value x doc-type => new-value`

**Description**

The generic function `documentation` operates as specified in the ANSI Common Lisp standard. Additional methods with signatures:

```lisp
(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:

```lisp
(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*

**Summary**
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 Note**
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
:hp-ux   HP-UX
: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, 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 Linux (64-bit), x86/x64 Solaris (64-bit) and Windows (64-bit).
:sparc   Images that run on SPARC architecture.
:powerpc Images that run on PowerPC architecture.
:hppa    Images that run in HP PA-RISC architecture.
:little-endian The compiler targets a little endian machine, for instance x86.
Code can distinguish the fourteen current LispWorks implementations like this:

```lisp
#+(and :mswindows :lispworks-32bit)
"LispWorks (32-bit) for Windows"
#+(and :mswindows :lispworks-64bit)
"LispWorks (64-bit) for Windows"
#+(and :linux :lispworks-32bit)
"LispWorks (32-bit) for Linux"
#+(and :linux :lispworks-64bit)
"LispWorks (64-bit) for Linux"
#+freebsd
"LispWorks for FreeBSD"
#+(and :darwin :x86)
"LispWorks (32-bit) for Macintosh (running on Intel)"
#+(and :darwin :powerpc :lispworks-32bit)
"LispWorks (32-bit) for Macintosh (running on PowerPC)"
#+(and :darwin :x86-64 :lispworks-64bit)
"LispWorks (64-bit) for Macintosh (running on Intel)"
#+(and :darwin :powerpc :lispworks-64bit)
"LispWorks (64-bit) for Macintosh (running on PowerPC)"
#+(and :solaris2 :x86)
"LispWorks (32-bit) for Intel/Solaris"
#+(and :solaris2 :x86-64)
"LispWorks (64-bit) for Intel/Solaris"
#+(and :sparc :lispworks-32bit)
"LispWorks (32-bit) for Solaris"
#+(and :sparc :lispworks-64bit)
"LispWorks (64-bit) for Solaris"
#+:hppa
"LispWorks for HP PA"
```

The following features can be used to distinguish between versions of LispWorks:

`:lispworks4` All major version 4 releases.

`:lispworks4.4` Release 4.4.x

`:lispworks5` All major version 5 releases.

`:lispworks5.0` Release 5.0.x

`:lispworks5.1` Release 5.1.x

`:lispworks6.0` Release 6.0.x
Every LispWorks 5 and LispWorks 6 image has exactly one of the features :lispworks-32bit and :lispworks-64bit.

For :sparc, :powerpc and :mswindows, there two LispWorks architectures: 32-bit and 64-bit, which can be distinguished by :lispworks-32bit or :lispworks-64bit.

The following features are present in LispWorks with the meanings defined for ANSI CL:

:ansi-cl
:common-lisp
:ieee-floating-point

Note that 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.

For a LispWorks image with the CAPI loaded, :capi will appear on *features*.

Note: 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.

**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.
### interactive-stream-p

**Function**

A generic function that determines if an object is an interactive stream.

<table>
<thead>
<tr>
<th>Package</th>
<th>cl</th>
</tr>
</thead>
</table>

#### Signature

`interactive-stream-p stream -> bool`

#### Arguments

- `stream`: A stream.

#### Values

- `bool`: 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`. There is an example in “Stream directionality” on page 270.

#### 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:

```lisp
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 Lisp-Works image.

See also: `short-site-name`

---

**loop**

*Macro*

Summary: A macro that performs iteration.

Package: cl

Signature:

```lisp
loop {for|as} var [type-spec]
being {the|each}{records|record}
{in|of} query-expression => result
```

Arguments:

- `var`: A variable.
- `query-expression`: An 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 38, “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 => new-array)
```

**Arguments**

- **weak**
  A generalized boolean.

- **allocation**
  A fixnum, or one of nil, :new, :static, :old, or :long-lived.

- **single-thread**
  A generalized boolean.

**Description**
The standard definition of `make-array` is extended to accept the keyword arguments :weak and :allocation.

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 signalled. 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.

- **:static**
  Allocate the array in a static segment.
Allocate the array assuming it is going to be long-lived.

Same meaning as :long-lived

Allocate the array in generation n.

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.

See also
array-weak-p
set-array-single-thread-p
set-array-weak

**make-hash-table**

*Function*

**Summary**

Creates and returns a new hash table which, in addition to the standard functionality, can have a user-defined test, a user-defined hash function, and be a weak hash table.

**Package**

common-lisp

**Signature**

make-hash-table &key test size rehash-size rehash-threshold hash-function weak-kind single-thread free-function => hash-table

**Arguments**

test A designator for a function of two arguments, which returns t if they should be regarded as the same and nil otherwise.
hash-function: A designator for a function of one argument, which returns a hash value.

weak-kind: One of :value, t, :key, :both, :one, :either, nil. The default is nil.

single-thread: A generalized boolean.

free-function: A designator for a function of two arguments.

**Description**

The standard definition of make-hash-table is extended such that test can be any suitable user-defined function, except that it must not call process-wait or similar mp package functions which suspend the current process. If test is not one of the standard test functions (eq, eql, equal and equalp), and if hash-function is not supplied, then the hash value is the same as would be used if test were equalp.

hash-function may be supplied only if test is not one of the standard test functions. It takes a hash key as its argument and returns a hash value to use for hashing.

If weak-kind is non-nil, it makes hash-table weak. Its semantics are the same as the second argument of set-hash-table-weak, that is:

```
(make-hash-table :weak-kind weak-kind <other-args>)
```

is equivalent to

```
(let ((ht (make-hash-table <other-args>))
   (set-hash-table-weak ht weak-kind)
   ht)
```

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 \texttt{nil}.

\texttt{free-function} adds a "free action" for a weak hash table. This has an effect only if \texttt{make-hash-table} is called with \texttt{weak-kind} non-nil. The \texttt{free-function} is called after an entry is automatically removed by the Garbage Collector. If \texttt{weak-kind} is \texttt{nil}, \texttt{free-function} is ignored.

\texttt{free-function}, if supplied, must take two arguments: \texttt{key} and \texttt{value}. When an entry is removed from a weak table \texttt{hash-table} because the relevant object is not pointed by any other object, the \texttt{key} and the \texttt{value} are remembered. Some time later (normally short, but not well-defined) the \texttt{free-function} is called with \texttt{key} and \texttt{value} as its arguments.

\texttt{free-function} needs to be fast, to avoid delays in unexpected places. Otherwise there are no restrictions on what \texttt{free-function} does. In particular, it can keep the \texttt{key} or \texttt{value} alive by storing them somewhere.

When objects are removed from the table by explicit calls (\texttt{remhash}, \texttt{clrhash}, (\texttt{setf gethash})), \texttt{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.

\texttt{free-function} can also be specified in a call to \texttt{set-hash-table-weak}.

**See also**

\texttt{modify-hash}
\texttt{set-hash-table-weak}
\texttt{with-hash-table-locked}
### make-instance

**Generic Function**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Creates and returns a new instance of a class.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>common-lisp</td>
</tr>
<tr>
<td>Signature</td>
<td>make-instance class &amp;rest initargs &amp;key &amp;allow-other-keys =&gt; instance</td>
</tr>
<tr>
<td>Arguments</td>
<td>class A class, or a symbol that names a class.</td>
</tr>
<tr>
<td></td>
<td>initargs An initialization argument list.</td>
</tr>
<tr>
<td>Values</td>
<td>instance A fresh instance of class class.</td>
</tr>
<tr>
<td>Description</td>
<td>make-instance behaves as specified in ANSI Common Lisp.</td>
</tr>
<tr>
<td></td>
<td>In particular it checks the initialization arguments as calculated by compute-class-potential-initargs.</td>
</tr>
<tr>
<td></td>
<td>This check can be suppressed by passing :allow-other-keys t. In addition, LispWorks provides global control over the initarg checking via set-make-instance-argument-checking and per-class control via class-extra-initargs.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> in a delivered image, make-instance does not check the initialization arguments.</td>
</tr>
<tr>
<td>Compatibility Note</td>
<td>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.</td>
</tr>
<tr>
<td>See also</td>
<td>class-extra-initargs</td>
</tr>
<tr>
<td></td>
<td>compute-class-potential-initargs</td>
</tr>
<tr>
<td></td>
<td>set-make-instance-argument-checking</td>
</tr>
</tbody>
</table>
**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 `make-sequence` function 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 signalled.

**See also**
- `concatenate`
- `map`
- `merge`

**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.

Description The `map` function 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 signalled.

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 `merge result-type sequence1 sequence2 predicate &key key => sequence`

Arguments `result-type`  A type specifier.
`sequence1`  A sequence.
`sequence2`  A sequence.
`predicate`  A designator for a function.
`key`  A designator for a function or `nil`.

Values `sequence`  A sequence.

Description The `merge` function 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 signalled.
Function

open

Summary
Creates, opens, and returns a file stream that is connected to a specified file.

Package
common-lisp

Signature
open filespec &key direction element-type external-format if-exists if-does-not-exist => stream

Arguments
filespec A file designator.
direction If direction is :probe, external-format is ignored. The element type and external format of the returned stream are undefined.
element-type By default, the value of *default-character-element-type* (the ANSI standard default is character).
external-format An external file format designator. By default, this is :default.
if-exists What to do if the file stream already exists. The possible values for this are as in the ANSI standard.
if-does-not-exist What to do if the file stream does not already exist. The possible values for this are as in the ANSI standard.

Values
stream A file stream, or nil.

Description
If external-format has a name which is not :default and the parameters include :eol-style, it is used as is.

See also
concatenate
make-sequence
map
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 (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 signalled 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 signalled.

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.
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.

See also
*default-character-element-type*
guess-external-format
set-file-dates
valid-external-format-p
### output-stream-p

**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`.

There is an example in “Stream directionality” on page 270.

**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.

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 88 for a more extended description of the compiler optimize qualities.

Examples

(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*

clause::= (case-name lambda-list [:interactive interactive-expression] | :report report-expression | :test test-expression] declaration* form*)

**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 `:default`.

**Values**
room returns no values.
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.

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 itself. The free space figure covers all the free space in all segments.

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 garbage collector 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.
Examples

CL-USER 23 > (room nil)
Total Size 50752K, Allocated 42868K, Free 7522K

CL-USER 24 > (room)
Generation 0: Total Size 2778K, Allocated 519K, Free 2251K
Generation 1: Total Size 3958K, Allocated 2524K, Free 1413K
Generation 2: Total Size 24324K, Allocated 20730K, Free 3572K
Generation 3: Total Size 19391K, Allocated 19266K, Free 112K
Total Size 50752K, Allocated 43040K, Free 7349K

CL-USER 25 > (room t)
Generation 0: Total Size 2778K, Allocated 561K, Free 2208K
  Segment 200877D8: Total Size 507K, Allocated 457K, Free 46K
    minimum free space 64K,
    Pending promotion = 1K, sweeps
before promotion = 10
  Segment 22F58548: Total Size 2270K, Allocated 104K, Free 2162K
    minimum free space 0K,
    Pending promotion = 0K, sweeps
before promotion = 2
  Generation 1: Total Size 3958K, Allocated 2524K, Free 1413K
  Segment 21C08548: Total Size 1472K, Allocated 1423K, Free 44K
    minimum free space 0K,
    Pending promotion = 0K, sweeps
before promotion = 4
  Segment 22198548: Total Size 1088K, Allocated 778K, Free 305K
    minimum free space 0K,
    Pending promotion = 0K, sweeps
before promotion = 4
  Segment 20706770: Total Size 68K, Allocated 3K, Free 65K
    minimum free space 3K,
    Pending promotion = 0K, sweeps
before promotion = 4
  Segment 216D8548: Total Size 1088K, Allocated...
213K, Free 870K
minimum free space OK,
Awaiting promotion = 0K, sweeps
before promotion = 4
Segment 2004AFA8: Total Size 242K, Allocated 105K, Free 132K
minimum free space 0K, static
Generation 2: Total Size 24324K, Allocated 20730K, Free 3572K
Segment 222A8548: Total Size 12992K, Allocated 9527K, Free 3460K
minimum free space 0K,
Awaiting promotion = 0K, sweeps
before promotion = 4
Segment 21D78548: Total Size 4224K, Allocated 4110K, Free 109K
minimum free space 0K,
Awaiting promotion = 0K, sweeps
before promotion = 4
Segment 21418548: Total Size 2816K, Allocated 2811K, Free 0K
minimum free space 0K,
Awaiting promotion = 0K, sweeps
before promotion = 4
Segment 217E8548: Total Size 4224K, Allocated 4218K, Free 1K
minimum free space 0K,
Awaiting promotion = 0K, sweeps
before promotion = 4
Segment 20DBDDC8: Total Size 68K, Allocated 63K, Free 0K
minimum free space 117K,
Awaiting promotion = 0K, sweeps
before promotion = 4
Generation 3: Total Size 19391K, Allocated 19266K, Free 112K
Segment 20106770: Total Size 6144K, Allocated 6139K, Free 0K
minimum free space 3K,
Awaiting promotion = 0K, sweeps
before promotion = 10
Segment 20DCEE40: Total Size 6437K, Allocated 6321K, Free 112K
minimum free space 0K,
Awaiting promotion = 0K, sweeps
before promotion = 10
Segment 207177E8: Total Size 6809K, Allocated
27 The COMMON-LISP Package

6805K, Free 0K
minimum free space 0K,
Awaiting promotion = 0K, sweeps
before promotion =10
Total Size 50752K, Allocated 43083K, Free 7307K

See also find-object-size
room-values
total-allocation

short-float

Type

Summary A subtype of float.

Package common-lisp

Signature short-float

Description A short float is an immediate object.

short-float is disjoint from double-float in all LispWorks implementations in version 5.0 and later.

short-float is disjoint from single-float in all 32-bit LispWorks implementations, version 5.0 and later. In 64-bit LispWorks short-float is the same type as single-float.

Compatibility Note

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

440
**short-site-name**

*Function*

**Summary**
Ids the physical location of the computer.

**Package**
common-lisp

**Signature**

```lisp
short-site-name => description
(setf short-site-name) description => description
```

**Arguments**

- `description`: A string or `nil`.

**Description**
The function `short-site-name` returns a string briefly identifying the physical location of the computer. This should be set using `(setf short-site-name)` when you configure your LispWorks image.

**See also**
long-site-name

---

**simple-base-string**

*Type*

**Summary**
The simple base string type.

**Package**
common-lisp

**Signature**

```lisp
simple-base-string length
```

**Arguments**

- `length`: The length of the string (or `*`, meaning any).

**Description**
The type of simple base strings.

---

**single-float**

*Type*

**Summary**
A subtype of `float`.

**Package**
common-lisp
### single-float

**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.

A `single-float` is disjoint from `double-float` in all LispWorks implementations, version 5.0 and later.

A `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 Note**
In LispWorks 4.4 and previous on Windows and Linux platforms, `single-float` is the same type as `double-float`. However, there are distinct specialized array types `(array single-float)`, with single precision, and `(array double-float)`, with double precision.

**See also**
- `double-float`
- `long-float`
- `parse-float`
- `short-float`

---

### software-type

**Function**

**Summary**
Identifies the Operating System.

**Package**
`common-lisp`

**Signature**
```
software-type => description
```

**Values**
- `description` A string or `nil`.

**Description**
The function `software-type` returns a string representing a generic name of the Operating System, or `nil` if this cannot be determined.

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.


**Example**

```
(software-version)  
=>  
"Windows Vista: 6.0 (build 6000) "

(software-version)  
=>  
"Windows XP: 5.1 (build 2600) Service Pack 2"
```
(software-version)
=>
"Windows Millennium: 4.90 (build 3000)"

See also software-type

**step**

Macro

Summary Steps through the evaluation of a form.

Package common-lisp

Signature `step form => result`

Arguments `form` A form to be stepped and evaluated.

Values `result` The values returned by `form`.

Description `step` evaluates a form and allows you to single-step through it. You can include a call to `step` inside a tricky definition to invoke the stepper every time the definition is used. `step` can also optionally step through macros.

The commands shown below are available. When certain stepper variables (as described below) are set, some of these commands are not relevant and are therefore not available. Use `:help` to get a list of the commands.

`:s n` Step this form and all of its subforms (optional positive integer argument).

`:st` Step this form without stepping its subforms.

`:su` Step up out of this form without stepping its subforms.

`:sr` Return a value to use for this form.

`:sq` Quit from the current stepper level.
:redo Redo one of the previous commands.
:get Get an item from the history list and put it in a variable.
:help List available commands.
:use Replace one form with another form in previous command and redo it.
:his List the commands history.

The optional integer argument $n$ for :s means do :s $n$ times.

**Note:** :step is a Listener-based form stepper. LispWorks also offers a graphical source-code Stepper tool. See the *LispWorks IDE User Guide* for details of that.

**Examples**

The following examples illustrate some of these commands.

```
USER 12 > (step (+ 1 (* 2 3) 4))
(+ 1 (* 2 3) 4) -> :s
  1 -> :s
  1
(* 2 3) -> :su
  6
  4 -> :s
  4
11
11

USER 13 > (defun foo (x y) (+ x y))
FOO

USER 14 > step (foo (+ 1 1) 2)
(FOO (+ 1 1) 2) -> :st
(+ 1 1) -> :s
  1 -> :s
  1
  1 -> :s
  1
  2
  2 -> :s
  2
  4
  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
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**

type A type specifier.

**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 implement-
tation 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 270.

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 *.

**Description**
The union of all string types as specified in the standard, but extended with an extra parameter: element-type.

(string length element-type) means all string types whose element type is a subtype of element-type. That is:
(string * base-char) = (vector base-char *)
(string * lw:simple-char) = (or (vector base-char *)
                   (vector lw:simple-char *))
(string * character) = (or (vector base-char *)
                     (vector lw:simple-char *)
                     (vector character *))

Example
CL-USER 235 > (lw:set-default-character-element-type 'base-char)
BASE-CHAR
CL-USER 236 > (concatenate 'string "f" "o" "o")
"foo"
CL-USER 237 > (type-of *)
SIMPLE-BASE-STRING

See also
*default-character-element-type*
set-default-character-element-type

**time**

*Macro*

Summary
Determines the execution time of a form in the current environment.

Package
common-lisp

Signature
time form => result

Arguments
form A form to be evaluated.

Values
result The result of the evaluation of the form.

Description
`time` can be used to determine execution times. The macro evaluates the form `form` and returns its value `result`. `time` also prints some timing and size data: `user time`, `system time`, `elapsed time`, and the total amount of heap space allocated in executing the form (in bytes).
The *user time* printed is the time used by LispWorks or any code that it calls in a dynamic library.

The *system time* printed is the time used in the operating system kernel when it is doing work on behalf of the LispWorks process.

The *elapsed time* printed is the time you could in principle measure with a stopwatch.

If LispWorks is 100% busy throughout the execution of the code, then *user time + system time = elapsed time*.

Each of the times is printed as:

- *secs.micros* if less than 60 seconds
- *hours:minutes:secs.micros* if 60 seconds or more.

The timing and size data covers all stack groups, not just the one that invokes *time*.

### Notes

1. Note that *time* itself uses a small, constant amount of heap space.

2. *time* measures all threads, so to test accurately for cons-ing in *code* you need to do:

   ```lisp
   (sys:with-other-threads-disabled (time code))
   ```

   This is particularly important when using the LispWorks IDE. Do not use `sys:with-unique-names` 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   = 4932022956 bytes
0 Page faults
Calls to %EVAL = 72000048
3.4606518E9
See also  
extended-time  
with-other-threads-disabled  
with-unique-names

trace

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)

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 special-
ized parameters of an existing method, and then only this method is traced (as opposed to the corresponding generic function).

\textbf{tracing-desc} \hspace{1em} Specifies the functional definition which is to be traced and specifies any additional options that are required.

\textbf{: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 \texttt{hcl:*traced-results*} then the values received by the caller of the function are correspondingly altered (see also \texttt{hcl:*traced-results*}).

\textbf{: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 \texttt{nil}), \texttt{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,

\begin{verbatim}
(trace (print :entrycond nil :exitcond nil :allocation $print-allocation))
\end{verbatim}

results in \$\texttt{print-allocation} containing the sum of the allocation made inside \texttt{print}.

Note that if the function is called again, \texttt{trace} continues to use \$\texttt{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.

\textbf{: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:

\textbf{:quick} \hspace{1em} Like the \texttt{:bq} debugger command.
Like the :b debugger command.

Like the :b :verbose debugger command.

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, i.e. :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  trace-result

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.

**Note:** trace works by tracing function names, not function objects. Therefore tracing function objects, for example by

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

will not yield any trace output. Also, if the symbol foo is traced, then invoking the function foo by

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

will not produce any trace output.

**Note:** for detailed information about the current tracing state, call tracing-state.
Example 1

USER 1 > (defvar *number-of-calls-to-max* 0)
*NUMBER-OF-CALLS-TO-MAX*
USER 2 > (trace (max :after
  ((incf *number-of-calls-to-max*))))
(MAX)
USER 3 > (dotimes (i 2) (max i 1))
  0 MAX > (0 1)
  0 MAX < (1)
  1
  0 MAX > (1 1)
  0 MAX < (1)
  2
NIL
USER 4 > *number-of-calls-to-max*
  2
USER 5 > (trace (max
  :entrycond
    (> (length compiler:*traced-arglist*)
    2)
  :exitcond nil))
(MAX)
USER 6 > (max 2 3 (max 4 5))
  0 MAX > (2 3 5)
  5

Example 2

This example illustrates the use of :inside.
Example 3
To trace a method:

```lisp
(defmethod foo (x) x)
(trace ((method foo (t))))
```

Example 4
To trace a setf function:

```lisp
(defun (setf foo) (x y) (set y x))
(trace (setf foo))
```

```lisp
(setf (foo '*a*) 42)
0 (SETF FOO) > (42 *A*)
>> X : 42
>> Y : *A*
0 (SETF FOO) < (42)
42
```
### 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.
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 Is 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
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 `compiler::%pass%`, in which case no transformation is applied).

**Values**

**list-of-transforms** A list of the names of transforms defined for the function, including the one just added.

**Description**

`deftransform`, like `defmacro`, defines a function that computes the expansion of a form. Transforms are only used by the compiler and not by the interpreter. `deftransform` allows you to add to the optimizations performed by the compiler.

**Examples**

```lisp
(compiler:deftransform  +  +of-2  (x y)
  '(system::|+2| ,x ,y))
(compiler:deftransform  +  +of-many (x &rest y)
  '(system::|+2| ,x (+ ,@y)))
;; now an expression like (+ a b c 4 5 7 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
(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
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)
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)
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**
*hidden-packages*
logs-directory
The DBG Package

*hidden-packages*  

**Variable**

**Summary**  
A list of packages whose symbols should not be displayed in debugger output.

**Package**  
dbg

**Initial Value**  
A list containing the dbg and conditions packages.

**Description**  
dbg:*hidden-packages* is used by the debugger. It should be bound to a list of package specifiers. If a package is included in the list then any symbols in it are not shown by the debugger. Thus during backtraces the call frames corresponding to functions in these packages are not displayed. This can be useful in restricting the debugger to particular areas.
Example

CL-USER 1 > unbound

Error: The variable UNBOUND is unbound.
1 (continue) Try evaluating UNBOUND again.
2 Return the value of :UNBOUND instead.
3 Specify a value to use this time instead of evaluating UNBOUND.
4 Specify a value to set UNBOUND to.
5 (abort) Return to level 0.
6 Return to top loop level 0.

Type :b for backtrace or :c <option number> to proceed.
Type :log-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

*hidden-packages* is an extension to Common Lisp.

log-bug-form

Function

Summary

Writes a log of an error. This is useful in an application’s error handlers.

Package

dbg
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*.
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`

**Function**

**Summary**
Returns the directory in which LispWorks puts log files.

**Package**
dbg

**Signature**

```lisp
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`

*hidden-packages*
**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**
### *print-binding-frames*  

<table>
<thead>
<tr>
<th><strong>Summary</strong></th>
<th>Controls whether binding frames are printed in debugger output.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package</strong></td>
<td>dbg</td>
</tr>
<tr>
<td><strong>Initial Value</strong></td>
<td>nil</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>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 <code>dbg:*print-binding-frames*</code> is true then the binding frames are shown.</td>
</tr>
</tbody>
</table>
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)
### Notes

*print-binding-frames* is an extension to Common Lisp.

### *print-catch-frames*

**Variable**

**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.
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 :

Notes

*print-catch-frames* is an extension to Common Lisp.

*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-han-
`print-handler-frames` is set to `t` then the handler frames are displayed.

**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 :
```

**Notes**

`*print-handler-frames*` is an extension to Common Lisp.

`print-open-frames`  

**Variable**

**Summary**

Controls whether open frames are printed in debugger output.
Package:  
\texttt{dbg}

Initial Value: \texttt{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 \texttt{*print-open-frames*} is set to \texttt{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:
\begin{verbatim}
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 
\end{verbatim}

Notes: \texttt{*print-open-frames*} is an extension to Common Lisp.

\texttt{*print-restart-frames*} 

Variable

Summary: Controls whether restart frames are printed in debugger output.

Package: \texttt{dbg}

Initial Value: \texttt{nil}

Description: This variable is used by the debugger when it displays the stack frames. Restart frames are formed when restarts are established (see “The stack in the debugger” on page 12), but
are normally not shown by the debugger. However if
*print-restart-frames* is set to t then the restart frames
are shown.

Example

USER 43 > (setq dbg:*print-restart-frames* t)
T
USER 44 > (truncate 12.5 0.0)
Error: Division-by-zero caused by TRUNCATE
of (12.5 0.0)
  1 (continue) Return a value to use
  2 Supply new arguments to use
  3 (abort) return to top loop level 0.
Type :c followed by a number to proceed
USER 45 : 1 > :b 5
Restart frame: (ABORT)
Catch frame: (NIL)
Catch frame: #:|block-catcher-3223|
Call to DIVISION-BY-ZERO-ERROR :
Call to TRUNCATEANY :
USER 46 : 1 >

Notes
*print-restart-frames* is an extension to Common Lisp.

*terminal-debugger-block-multiprocessing*

Variable

Summary Controls blocking of multiprocessing in the terminal debug-
er.

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  
(setf dbg:*terminal-debugger-block-multiprocessing* nil)
NIL

CL-USER 7 > unbound

Error: The variable UNBOUND is unbound.
1 (continue) Try evaluating UNBOUND again.
2 Specify a value to use this time instead of evaluating UNBOUND.
3 Specify a value to set UNBOUND to.
4 (abort) Return to level 0.
5 Return to top-level loop.
6 Return from multiprocessing.

Type :b for backtrace, :c <option number> to proceed, or :? for other options

CL-USER 8 : 1 > (setq *timer* (mp:make-timer 'print 10))
#<Time Event : PRINT>

CL-USER 9 : 1 > (mp:schedule-timer-relative *timer* 1)
#<Time Event : PRINT>

CL-USER 10 : 1 >
10

Notice above that the output 10 from the call to print appears after 1 second, in the debugger. Multiprocessing was not blocked.

See also start-tty-listener

with-debugger-wrapper  

Macro

Summary  
Executes code with a “debugger wrapper” which is called only if the debugger is invoked during the execution.

Package  
dbg
### Signature

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>with-debugger-wrapper</code></td>
<td><code>wrapper &amp;body body =&gt; results</code></td>
</tr>
</tbody>
</table>

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wrapper</code></td>
<td>A function designator.</td>
</tr>
<tr>
<td><code>body</code></td>
<td>Forms.</td>
</tr>
</tbody>
</table>

### Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>results</code></td>
<td>Results of <code>body</code>.</td>
</tr>
</tbody>
</table>

### 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:

```lisp
(funcall function condition)
```

### Example

Suppose that you run many processes in parallel with the same code. If the code is broken then every process will get an error. This example shows how a debugger wrapper can be used to keep a lock around entry to the debugger, so that the processes enter the debugger one by one. It contains firstly the "application code", then the debugger wrapper, and lastly forms which execute the application with or without the debugger wrapper.
(in-package "CL-USER")

(defglobal-parameter *a* 0)

(defun foo (index cons)
  (sys:atomic-push (* index *a*) (cdr cons)))

;; This gets the process function so we can pass
;; the wrapper function instead.
(defun my-run-processes (do-error &optional (process-function 'foo))
  (setq *a* (if do-error :do-error 7))
  (let ((cons (cons nil nil)))
    (dotimes (x 10)
      (mp:process-run-function
       (format nil "My test process -d" x)
       ()
       process-function
       x cons))
    (sleep 0.2)
    (print (cdr cons))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ; ; debugger wrapper ; ; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(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)
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**

**Variable**

**Summary**

Controls how source finding operates.

**Package**

dspec

**Initial Value**

(:internal)

**Description**

The *active-finders* variable 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.

See also
- `discard-source-info`
- `find-dspec-locations`
- `find-name-locations`
- `save-tags-database`

---

**at-location**

*Macro*

**Summary**

Tells the dspec system of the source location.

**Package**

dspec

**Signature**

`at-location (location) \&body body => result`

**Arguments**

- `location`: A pathname or a keyword.
- `body`: Forms, including defining forms.

**Values**

- `result`: The result of `body`.

**Description**

The macro `at-location` informs the dspec system that the source for definitions done during the execution of `body` are at the location `location`. 
location is usually a pathname, for definitions occurring in a file or editor buffer with that pathname.

Other locations are reserved for internal use. These are:

An editor buffer Defined in an editor buffer with no pathname.

:listener Interactively defined.

:unknown Defined without dspec information being recorded.

:implicit An aggregate defined by the existence of a part.

(:inside dspec loc)

A subform of dspec at location loc.

---

canonicalize-dspec  

**Function**

**Summary**

Returns the canonical form for a dspec.

**Package**

dspeg

**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**

dsdep

**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.
Description

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

Summary

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, pro-
viding 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 omit-
ted if not needed. The most important arguments for the
LispWorks IDE are definedp and undefiner.

If undefiner is given, its value must be a function of one argu-
ment. When LispWorks wants to remove a definition, it will
call the function with a canonical dspec of class name. The
function should returns a form that removes the current defi-
nition 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 given, 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 function returns the dspec if it matches the form

```
(dspec-class symbol)
```

If `prettify` is given, its value must be a function of one argument. 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 given, its value must be function of one argument. 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` otherwise. The default `definedp` function always returns `nil`.

If `object-dspec` is given, its value must be a function of one argument. When LispWorks wants to find the dspec that created 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` otherwise. For example, the `object-dspec` function for package dspecs might be:

```lisp
#'(lambda (obj)
    (and (packagep obj)
        `(package ,(package-name obj))))
```

The `object-dspec` function is used by the "Find Source" menu option in the Inspector in the LispWorks IDE to find where the current object was defined.
If `defined-parts` is given, 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 dspecs which are defined parts of the primary name in the class `name`. If this keyword is given, `aggregate-class` must also be given.

If `aggregate-class` is given, its value must be a symbol naming a dspec class that is the aggregate class of the parts defined by `name` dspecs. For example, the aggregate class of `method` is `defgeneric` because methods are the defined parts of a particular generic function. If this keyword is given, the `defined-parts` must also be given.

To make `cl:documentation` work for your dspec class, add a suitable method as described for `documentation`.

Example
See “Dspec classes” on page 63.

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**
```
define-form-parser definer-and-options &optional parameters
&body body => parser
```

**Arguments**
`definer-and-options`
A symbol `definer` naming a definer of functions, macros, variables and so on, or a list (`definer options`) where `options` is a plist of keys and values.

### Parameters

- **nil**, or list of parameters `params` in the top level form, optionally ending with `&rest param-getter`.

### Body

The body of a parser function.

### Values

- **parser**: A form parser function.

### Description

The macro `define-form-parser` defines a form parser for forms beginning with `definer`.

`options` is a property list with the following keys allowed:

- **:parser**: A parser function `parser-function`.
- **:alias**: A dspec class or alias `alias`.
- **:anonymous**: A boolean.

The parser function defined is named by `parser-function`. If the `:parser` option is omitted then the name defaults to a symbol in the current package whose symbol name is the symbol name of `definer` with `"-FORM-PARSER"` appended.

If `parameters` and `body` are given, then `parser-function` is defined as a global function that is expected to return a dspec for the defining form or `nil` if this is not possible. Within `body`, `definer` is bound to the `car` of the actual form being parsed. In simple cases, this is just `definer`, but if the form parser is used as in the `:alias` option of another form parser then the symbol will be bound to the `car` of that form instead.

The `params` are bound to subsequent subforms of the defining form. If `&rest param-getter` is supplied, then it is bound to a function of no arguments that returns two values: the next subform if there is one and a boolean to indicate if a subform was found.
If parameters and body are omitted, then parser-function is expected
to be a form parser defined by a different
define-form-parser form, or you can specify as an alias a definer with an existing form parser via the value alias of the :alias key in options.

If the :anonymous option is non-nil then definer is not associated with the form parser. This is useful in conjunction with parameters and body for defining generic form parsers that can be used in other define-form-parser forms.

LispWorks contains pre-defined form parser functions for the Common Lisp definers defun, defmethod, defgeneric, defvar, defparameter, defconstant, defstruct, defclass, defmacro and deftype and for LispWorks definers such as fli:define-foreign-type and dspec:define-form-parser itself.

When a defining symbol definer has an associated form parser, this parser function is used by the source location commands such as Expression > Find Source in the LispWorks IDE. Having identified the file where the definition was recorded, LispWorks parses the top level forms in the file looking for the one which matches the definition spec. When found, this match is displayed.

Example

Define a parser for def-bar forms whose name is made from the second element of the form and any subsequent keywords:

(dsmap:define-form-parser def-bar
  (def-bar (:parser def-bar-form-parser)))
The DSPEC Package

(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 (twonames (:anonymous t)) (name1 name2)
  `(,twonames ,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)))

See also get-form-parser
parse-form-dspec
**dspec-class**  
*Function*

Summary  Returns the dspec class of a dspec.

Package  **dspec**

Signature  

\[
\text{dspec-class dspec} \rightarrow \text{class}
\]

Arguments  

- **dspec**  A dspec.

Values  

- **class**  A dspec class name.

Description  The function **dspec-class** returns the dspec class name for **dspec**.

Example  

```
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-classes**  
*Variable*

Summary  Lists all the dspec classes.

Package  **dspec**

Signature  

\[
*\text{dspec-classes}*
\]

Description  The variable **dspec-classes** contains a list of the names of all the dspec classes.
**dspec-defined-p**

*Function*

**Summary**
The predicate for whether a dspec has a definition.

**Package**
dspec

**Signature**
dspec-defined-p  dspec  =>  definedp

**Arguments**
dspec  A dspec.

**Values**
definedp  The canonical form of dspec if dspec is defined, or nil otherwise.

**Description**
The function dspec-defined-p determines whether the dspec dspec has a definition. If so, it returns the canonical form of dspec.

If dspec has no definitions, dspec-defined-p returns nil.

**Example**
CL-USER 23 > (dspec:dspec-defined-p '(function list))
(DEFUN LIST)

---

**dspec-definition-locations**

*Function*

**Summary**
Returns the locations of the known definitions.

**Package**
dspec

**Signature**
dspec-definition-locations  dspec  =>  locations

**Arguments**
dspec  A dspec.

**Values**
locations  A list of pairs (recorded-dspec location).

**Description**
The function dspec-definition-locations returns the locations of the definitions recorded for the dspec dspec.
For each known definition recorded-dspec names the definition that defined dspec in location, and location is a pathname or keyword as described in at-location.

Note that non-file locations, such as :unknown, can occur in the list. The locations in locations are all basic locations: that is, there are no (:inside ...) locations.

If dspec is a local dspec, the parent function is located.

Example

```
CL-USER 6 > (dspec:dspec-definition-locations
'(defun foo-bar))
(((DEFSTRUCT FOO) #P"C:/temp/hack.lisp")
```

See also

name-definition-locations

**dspec-equal**

*Function*

**Summary**
Tests two dspecs for equality as dspecs.

**Package**
dspec

**Signature**
dspec-equal dspec1 dspec2 => result

**Arguments**
dspec1, dspec2 Dspecs.

**Values**
result A boolean.

**Description**
The function dspec-equal compares dspec1 and dspec2 for equality as dspecs.

Both arguments are canonicalized before the comparison.

Dspecs in different subclasses of the same namespace are dspec-equal if their names match.

Unknown dspecs are compared simply by equal.
Example

CL-USER 44 > (dspec:dspec-equal '((deftype foo) '(defclass foo)))
T

dspec-name

Function

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.
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**

```lisp
(dspec-progenitor
  '(subfunction 1 (subfunction (flet a) (defun foo))))
=>
  (defun foo)
```

See also: `local-dspec-p`
**dspec-subclass-p**

*Function*

Summary  Tests whether one dspec class is a subclass of another.

Package  dspec

Signature  dspec-subclass-p class1 class2 => result

Arguments  class1, class2  Symbols naming dspec classes.

Values  result  A boolean.

Description  The function dspec-subclass-p determines whether the dspec class denoted by class1 is a subclass of that denoted by class2.

Example  

```
CL-USER 55 > (dspec:dspec-subclass-p 'defmacro 'type)
NIL

CL-USER 56 > (dspec:dspec-subclass-p 'defmacro 'function)
T
```

**dspec-undefiner**

*Function*

Summary  Returns an undefining expression for a dspec.

Package  dspec

Signature  dspec-undefiner dspec => form

Arguments  dspec  A dspec.

Values  form  A Lisp form.

Description  The function dspec-undefiner returns a form which would undefine dspec, whether or not dspec is currently defined.
If no such form can be constructed, `nil` is returned.

**Example**

```
CL-USER 66 > (dspec:dspec-undefiner '(defun foo))
(PROGN (FMAKUNBOUND (QUOTE FOO)) (SETF (DOCUMENTATION (QUOTE FOO) (QUOTE FUNCTION)) NIL))
```

---

### `discard-source-info`  
*Function*

**Summary**
Cleans the internal dspec database.

**Package**
dspec

**Signature**
discard-source-info => nil

**Arguments**
None.

**Values**
Returns `nil`.

**Description**
The function `discard-source-info` removes all source location information from the internal dspec database.

**Example**
To build `my-image` which does not contain source locations for the definitions loaded, but retaining a tags database of those definitions:

```
(load-all-patches)
(load "my-code")
(dspec:save-tags-database #P"my-tags-database.ofasl")
(dspec:discard-source-info)
(save-image "my-image")
```

**See also**
save-tags-database

---

### `find-dspec-locations`  
*Function*

**Summary**
Returns the locations of the definitions of a dspec.

---
The function `find-dspec-locations` returns the locations of the relevant definitions for the dspec `dspec`.

For each known definition `recorded-dspec` names the definition that defined `dspec` in `location`, and `location` is a pathname or keyword as described in `at-location`.

If `dspec` is a local dspec, the parent function is located.

The location information is collected from all finders on *active-finders*, that is, the relevant definitions are those known to at least one of these finders.

If two or more finders return the same pair `(recorded-dspec location)`, as compared by `dspec-equal` and location equality, then only the first occurrence of the pair (in the order of *active-finders*) appears in `locations`.

See also
- *active-finders*
- dspec-definition-locations
- dspec-equal

---

**find-name-locations**

**Summary**

Returns the locations of the definitions of a name.

**Package**

dspec

**Signature**

`find-name-locations classes name => locations`
Arguments

classes  A list of dspec class names.
name  A name.

Values

locations  A list of pairs (recorded-dspec location).

Description

The function find-name-locations returns the locations of the relevant definitions for name in the classes listed in classes.

For each known definition recorded-dspec names the definition that defined name in location, and location is a pathname or keyword as described in at-location.

The location information is collected from all finders on *active-finders*, that is, the relevant definitions are those known to at least one of these finders.

If two or more finders return the same pair (recorded-dspec location), as compared by dspec-equal and location equality, then only the first occurrence of the pair (in the order of *active-finders*) appears in locations.

See also

*active-finders*
name-definition-locations
dspec-equal

get-form-parser  Function

Summary

Returns the form parser associated with a definer.

Package  dspec

Signature

get-form-parser definer => parser

Arguments

definer  A symbol naming a definer.

Values

parser  A form parser function, or nil.
**Description**

The function `get-form-parser` returns a form parser function if there is one associated with `definer`.

This is the case for predefined definers and for those for which you have established a form parser using `define-form-parser`.

If there is no associated form parser, `nil` is returned.

**Example**

```
CL-USER 1 > dspec:get-form-parser 'defun
DSPEC:NAME-ONLY-FORM-PARSER
```

**See also**

`define-form-parser`

`parse-form-dspec`

---

**local-dspec-p**

**Function**

**Summary**

The predicate for local dspecs.

**Package**

dspec

**Signature**

`local-dspec-p dspec => localp`

**Arguments**

`dspec` A dspec.

**Values**

`localp` A boolean.

**Description**

The function `local-dspec-p` determines whether the dspec `dspec` is a local dspec.

Local dspecs name local definitions, such as local functions.

Currently a local dspec is a list whose `car` is `subfunction`.

**See also**

`dspec-progenitor`
location

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
(defun define-wibble (x y)
  `(dspec:def (define-wibble ,x)
    (set-wibble-definition ',x ',y (dspec:location))))

(defun set-wibble-definition (x y loc)
  (when (record-definition `(define-wibble ,x) loc)
    ;; defining code here
  ))

See also
at-location
def

name-defined-dspecs

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 locations 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-finders*.
Example

CL-USER 7 > (dspec:name-definition-locations '(function) 'foo-bar)
(((DEFSTRUCT FOO) #P"C:/temp/hack.lisp")

See also

dspec-definition-locations

name-only-form-parser

Function

Summary

A pre-defined form parser.

Package

dspec

Signature

name-only-form-parser top-level-form getter => dspec

Arguments

top-level-form A top level defining form.
getter The subform getter function.

Values

dspec A dspec.

Description

The function name-only-form-parser is a predefined form parser for use with define-form-parser. The parser consumes one subform and returns it.

name-only-form-parser can be used for function definitions where the function name is an abbreviation for the full dspec. It is the predefined parser for defun, defmacro and defgeneric forms.

You can define it to be the parser for your defining forms. using define-form-parser.

Example

(defmacro my-definer (name &body body)
  `(defun ,name (x)
     ,@body))

(dspec:define-form-parser
  (my-definer (:parser
                  dspec:name-only-form-parser)))
See also  define-form-parser

**parse-form-dspec**

*Function*

**Summary**
Parses the dspec from a defining form.

**Package**
ds pec

**Signature**
parse-form-dspec form => result

**Arguments**
form A form.

**Values**
result A dspec or nil.

**Description**
The function `parse-form-dspec` invokes the defined form parser for `form` and returns the resulting dspec.

**Example**

```
(parse-form-dspec '(def-foo my-foo (arg) (foo-it arg)))
```

=>

```
(def-foo my-foo)
```

**See also**
define-form-parser
g et-form-parser

**record-definition**

*Function*

**Summary**
Checks for existing definitions and records a new definition.

**Package**
ds pec

**Signature**
record-definition dspec location &key check-redefinition-p => result

**Arguments**
dspec A dspec.
location A pathname or keyword.
check-redefinition-p

A boolean.

<table>
<thead>
<tr>
<th>Values</th>
<th>result</th>
<th>A generalised boolean.</th>
</tr>
</thead>
</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.

Note: 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 note

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
*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 note**
*record-source-files* 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**
*active-finders*

---

*reredefinition-action*  
**Variable**

**Summary**
Specifies the action on some redefinitions.

**Package**
dspec

**Initial value**
:warn

**Description**
*reredefinition-action* controls messages about redefinitions seen by the source location system.

If *reredefinition-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 note  *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

**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 save-tags-database function 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*

**single-form-form-parser**  
*Function*

Summary  A pre-defined form parser.

Package  dspec
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`, `define-package`, `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`
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.

See also  define-form-parser

**traceable-dspec-p**  

*Function*

**Summary**  Tests whether definition can be traced.

**Package**  dspec

**Signature**  

traceable-dspec-p  


dspec  =>  result

**Arguments**  

dspec  A dspec.

**Values**  

result  A generalised boolean.
Description

The function `traceable-dspec-p` determines whether the `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

```lisp
CL-USER 67 > (dspec:traceable-dspec-p '(subfunction foo bar))
NIL

CL-USER 68 > (dspec:traceable-dspec-p '(defun open))
OPEN
```

### tracing-enabled-p

**Function**

**Summary**

Gets and sets the global tracing state.

**Package**

`dspec`

**Signature**

`tracing-enabled-p` => `enabledp`

`(setf tracing-enabled-p)` `enabledp` => `enabledp`

**Values**

`enabledp` A generalized boolean.

**Description**

The function `tracing-enabled-p` determines whether tracing (by the Common Lisp macro `trace`) is currently on. This is independent of whether any functions are currently traced.

The function `(setf tracing-enabled-p)` switches tracing on or off according to the value of `enabledp`. This does not affect the list of functions that are currently traced.

**See also**

`trace`

`tracing-state`
**tracing-state**

*Function*

**Summary**
Gets the current trace details.

**Package**
dspec

**Signature**
\[
\text{tracing-state} \ \&\text{optional} \ dspec \Rightarrow state
\]

**Signature**
\[
\text{(setf tracing-state)} \ state \ \&\text{optional} \ dspec \Rightarrow state
\]

**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 `\text{(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.

`\text{(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 DSPEC Package
This chapter describes symbols available in the **EXTERNAL-FORMAT** package. Use of these symbols are discussed in Chapter 22, “Internationalization”.

### char-external-code

| **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

\[ (+ (* 100 \text{ row}) \text{ column}) \]

: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

**find-external-char**

---

**decode-external-string**

*Function*

**Summary**

Decodes a binary vector to make a string.

**Package**

external-format

**Signature**

\[
\text{decode-external-string} \ \text{vector} \ \text{external-format} \ \&key \ \text{start} \ \text{end} \\
\Rightarrow \ \text{string}
\]

**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 note
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`. 
The EXTERNAL-FORMAT Package

Compatibility

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

temporary-function

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

Initargs  :name  The name of the external format involved.

Description  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.

temporary-function

external-format-foreign-type  Function

Summary  Returns a type specifier for the integers handled by a specified external format.

Package  external-format

Signature  external-format-foreign-type external-format => type-specifier

Arguments  external-format  An external character format.
Values: *type-specifier* A type specifier describing the integer types handled by *external-format*.

Description: Takes the name of an external format, and returns a Lisp type specifier for the type of integers that the external format handles on the foreign side.

See also: *external-format-type*

---

**external-format-type**  
*Function*

Summary: Returns a type specifier for the characters handled by a specified external format.

Package: *external-format*

Signature: `external-format-type external-format => type-specifier`

Arguments: *external-format* An external character format.

Values: *type-specifier* A type specifier describing the character types handled by *external-format*.

Description: Takes the name of an external format, and returns a type specifier for the type of characters that the external format handles on the Lisp side.

See also: *external-format-foreign-type*

---

**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 row) column)`

`:euc-jp` is the complete two-byte format encoded as

`(+ (* 256 first-byte) second-byte)`

`:sjis` is Shift-JIS encoded in the same way. Strictly speaking, EUC and Shift-JIS are not coded character sets, but encodings of the JIS sets, but the encoding is easily expressed as an integer, so the same interface to it is used.

See also `char-external-code`

**Function**

valid-external-format-p

Summary: Tests whether an external format spec is valid.
Package  external-format

Signature  valid-external-format-p ef-spec &optional env => bool

Arguments  ef-spec An external format spec.

env An environment across which the spec should apply.

Values  bool t if ef-spec is a valid spec; nil otherwise.

Description This predicate tests whether the external format spec given in ef-spec is valid (in the environment env).

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-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 symbol naming 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.

The functions `flag-special-free-action` and `flag-not-special-free-action` flag and unflag objects for action.

Example:

```
(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**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Adds a symbol to the list of profiled symbols.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td><code>hcl</code></td>
</tr>
<tr>
<td>Signature</td>
<td><code>add-symbol-profiler symbol =&gt; nil</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>symbol</code> A symbol to be added to the <code>*profile-symbol-list*</code>.</td>
</tr>
<tr>
<td>Values</td>
<td>Returns <code>nil</code>.</td>
</tr>
<tr>
<td>Description</td>
<td><code>add-symbol-profiler</code> adds a symbol to <code>*profile-symbol-list*</code>, the list of profiled symbols.</td>
</tr>
</tbody>
</table>
| 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.

**Note:** this macro 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*

function

**analysing-special-variables-usage**

**Summary**

Prints an analysis of proclaimed symbols seen during compilation, as an aid to improving declarations.

**Package**

hcl

**Signature**

```lisp
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.
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 98.

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

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.
avoid-gc

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.

Note: 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
**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.

**Note:** The cross-referencing information used by binds-who is generated when code is compiled with source-level debugging switched on.

**See also**
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*.
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, `(mark-and-sweep 2)` will free a large amount of space in generation 2. `block-promotion` can be thought of as doing `set-promotion-count` on generation 1 with an infinite `count`, for the duration of `body`.

`block-promotion` is suitable only for use in particular operations that are known to create such relatively long-lived, but transient, objects. In typical uses these are objects that live for a few seconds to several hours. An example usage is Lisp-Works `compile-file`, to ensure the transient compile-time data gets collected.

`block-promotion` has global scope and hence may not be useful in an application such as a multi-threaded server. During the execution of `body`, generation 1 grows to accommodate all the allocated data, which may have some negative effects on the behavior of the system, in particular on its interactive response.

**Note:** symbols and process stacks are allocated in generation 2 or 3 (see `*symbol-alloc-gen-num*) hence `block-promotion` cannot prevent these getting into that generation. `allocation-in-gen-num` can also cause allocation in higher generations.

**Note:** in 64-bit LispWorks, `block-promotion` is implemented using `set-blocking-gen-num`.

---

**See also**

- `allocation-in-gen-num`
- `mark-and-sweep`
- `set-promotion-count`
**building-universal-intermediate-p**

*Function*

**Summary**

Used in a build script to determine if LispWorks is building an intermediate image when making a universal binary.

**Package**

hcl

**Signature**

building-universal-intermediate-p => intermediatep

**Arguments**

None

**Values**

intermediatep   A boolean.

**Description**

The function `building-universal-intermediate-p` can be used in a build script to determine if it is being executed to build one of the architectures of a universal binary.

The return value `intermediatep` is `nil` in most cases. It will be `t` only when building an intermediate image for the purpose of building a universal binary, either by `save-universal-from-script` or the Application Builder (see the LispWorks IDE User Guide).

This is useful if there are some configuration that should be done only in a universal binary image but not in a mono-architecture ("thin") image. Whether the intermediate image will be the Intel or the PowerPC part of the universal binary can be determined by checking `*features*`.

On architectures that do not have universal binaries, this function always returns `nil`.

**See also**

`save-universal-from-script`

`save-argument-real-p`

**calls-who**

*Function*

**Summary**

Lists functions called by a function.
**calls-who**

**Signature**

\[ \text{calls-who } \text{dspec} \Rightarrow \text{callees} \]

**Arguments**

- **dspec**: A dspec.

**Values**

- **callees**: A list.

**Description**

The function `calls-who` returns a list of the dspecs naming the functions called by the function named by `dspec`.

See also the editor commands **List Callees**, and **Show Paths From**.

**Note:** 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**

\[ \text{cd } \text{&optional } \text{directory} \Rightarrow \text{current-dir} \]

**Arguments**

- **directory**: A pathname designator specifying the new directory.

**Values**

- **current-dir**: A physical pathname.
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 `"~/"`.

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.

See also `cd`  
`get-working-directory`

---

**check-fragmentation**

**Function**

**Summary**
Provides information about the fragmentation in a generation in 32-bit LispWorks.

**Package**
hcl
The HCL Package

Signature  
\texttt{check-fragmentation gen-num \Rightarrow total-free, total-small-blocks, total-large-blocks}

Arguments  
\texttt{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  
\texttt{total-free}  
Total free space in the generation.
\texttt{total-small-blocks}  
Amount of free space in the generation which is available in blocks of 512 bytes or larger.
\texttt{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 \texttt{try-move-in-generation}.

Note: \texttt{check-fragmentation} is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations, where \texttt{gen-num-segments-fragmentation-state} is available instead.

See also  
\texttt{try-compact-in-generation}
\texttt{try-move-in-generation}

\textbf{clean-down}  
\textit{Function}

Summary  
Frees memory and reduces the size of the image, if possible.

Package  
hcl

Signature  
\texttt{clean-down \&optional full \Rightarrow new-size}
Arguments

*full* controls whether to operate on the highest generation. The default is *t*.

Values

*new-size* The new size of the image, after reduction.

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.

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* may fail to delete empty segments if there are static segments in high address space.

Notes

*try-move-in-generation* uses less CPU than *clean-down*, though it does not do the mark and sweep.

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.

See also

*gc-generation*

*save-image*

*try-move-in-generation*
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.

**Note:** The function may not be very useful, as it may be more efficient to directly allocate the objects in a particular generation in the first place, using \texttt{allocation-in-gen-num} or \texttt{set-default-generation}.

**Note:** \texttt{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 \texttt{(gc-generation 0)}.

**Example**  
\begin{verbatim}
; allocate lots of non-ephemeral objects
; .......
(clean-generation-0)
\end{verbatim}

**See also**
- allocation-in-gen-num
- collect-generation-2
- collect-highest-generation
- expand-generation-1
- gc-generation
- set-promotion-count
**collect-generation-2**

*Function*

**Summary**
Controls whether generation 2 is garbage collected in 32-bit LispWorks.

**Package**
hcl

**Signature**
collect-generation-2 on => size

**Arguments**
on
If on is nil, generation 2 is not garbage collected. If on is t, the generation is garbage collected.

**Values**
size
The current size of the image.

**Description**
Controls whether generation 2 is garbage collected. (Generation 2 normally holds long-lived objects created dynamically.)

*Note:* 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

**collect-highest-generation**

*Function*

**Summary**
Controls whether the top generation is garbage-collected in 32-bit LispWorks.

**Package**
hcl
## The HCL Package

### collect-highest-generation flag

**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**

Note: 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

### *compiler-break-on-error*

**Variable**

**Summary**

Controls whether compile-file handles compilation errors.

**Package**

hcl

**Initial Value**

nil

**Description**

If an error occurs during compilation of a form by compile-file, an error handler normally causes the compilation of that form to be skipped, and the error is reported later.

When *compiler-break-on-error* is non-nil, an error during compilation by compile-file is signaled and the debugger is entered.

**See also**

compile-file
**compile-file-if-needed**  

**Function**

**Summary**  
Compiles a Lisp source file if it is newer than the corresponding fasl file.

**Package**  
hcl

**Signature**  
```
compile-file-if-needed input-pathname &key output-file load
&allow-other-keys => output-truename, warnings-p, failure-p
```

**Arguments**  
- **input-pathname**  A pathname designator.
- **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

```lisp
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

See also  compile-file
```

**copy-to-weak-simple-vector**

*Function*

**Summary**

Creates a weak vector with the same contents as the supplied vector.

**Package**

hcl

**Signature**

`copy-to-weak-simple-vector vector-t => weak-vector`

**Arguments**

`vector-t`  
An array of type `(vector t)`.

**Values**

`weak-vector`  
A weak array of type `(vector t)`. 
### Description

The function `copy-to-weak-simple-vector` creates and returns a weak vector with the same contents as the argument `vector-t`.

Apart from the checking of arguments, this is equivalent to:

```
(replace (make-array (length vector-t) :weak t) vector-t)
```

See `set-array-weak` for a description of weak vectors.

**See also**

- `make-array`
- `set-array-weak`

---

### create-macos-application-bundle

**Function**

**Summary**

Creates a Mac OS X application bundle for the running LispWorks image.

**Package**

`hcl`

**Signature**


**Arguments**

- `target-path` A pathname designator.
- `template-bundle` A pathname designator.
- `bundle-name` A string.
- `signature` A string.
- `package-type` A string.
- `extension` A string.
- `application-icns` A pathname designator.
- `identifier` A string.
values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values

Values
version is the version value, CFBundleVersion. If template-bundle is nil, version defaults to the value returned by cl:lisp-implementation-version.

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.

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".

The default value of document-types is t, which means copy them from template-bundle.

create-macos-application-bundle is implemented only in LispWorks for Macintosh.

See also save-image-with-bundle

create-universal-binary

Function

Summary Creates a universal binary from two mono-architecture LispWorks images.

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

This function is intended for advanced use. See the function `save-universal-from-script` for a simpler way to create a universal binary.

The function `create-universal-binary` writes a universal binary to the file `target-image` from the saved image files `src-image1` and `src-image2`. The value of `target-image` is returned.

The source images `src-image1` and `src-image2` must both be LispWorks for Macintosh mono-architecture ("thin") images and one should be for the Intel architecture and the other for the PowerPC architecture (the order is immaterial). For example, they could have been created by `save-image` or `deliver`.

**Note:** The function `create-universal-binary` checks that `src-image1` and `src-image2` are LispWorks images of different architectures, but it does not check how they were saved or how similar they are. You need to ensure that both images contain the same functionality.

**Note:** The function `create-universal-binary` can only be called from a LispWorks for Macintosh image that is itself a universal binary, such as the distributed image.

Example

Suppose that you have saved two images, `my-application-intel` and `my-application-powerpc`, which contains the same application code loaded on an Intel Macintosh and a PowerPC Macintosh. The following command will combine them into a universal binary `my-application` that will run on both kinds of Macintosh:

```lisp
(create-universal-binary "my-application"
  "my-application-intel"
  "my-application-powerpc")
```

See also

- `save-image`
- `save-universal-from-script`
**current-stack-length**

*Function*

Summary

Returns the size of the current stack.

Package

hcl

Signature

current-stack-length => stack-size

Arguments

None

Values

*stack-size*

The current size of the stack, in 32 bit words (in 32-bit implementations) or 64-bit words (in 64-bit implementations).

Compatibility note

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*

**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*

**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*

**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** 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**

---

**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**

```lisp
do-profiling &key initialize processes profile-waiting ignore-in-foreign sleep function arguments func-and-args print stream
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>initialize</code></td>
<td>A boolean.</td>
</tr>
<tr>
<td><code>processes</code></td>
<td>One of <code>:current</code>, <code>:all</code>, a <code>mp:process</code> or a list of <code>mp:process</code> objects.</td>
</tr>
<tr>
<td><code>profile-waiting</code></td>
<td>A boolean.</td>
</tr>
<tr>
<td><code>ignore-in-foreign</code></td>
<td>A boolean.</td>
</tr>
<tr>
<td><code>sleep</code></td>
<td>A non-negative number, or <code>nil</code>.</td>
</tr>
<tr>
<td><code>function</code></td>
<td>A function designator.</td>
</tr>
<tr>
<td><code>arguments</code></td>
<td>Arguments passed to <code>function</code>.</td>
</tr>
<tr>
<td><code>func-and-args</code></td>
<td>A function designator or a list (<code>function-designator . args</code>).</td>
</tr>
<tr>
<td><code>print</code></td>
<td>A generalized boolean.</td>
</tr>
<tr>
<td><code>stream</code></td>
<td>An output stream.</td>
</tr>
</tbody>
</table>

**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:

```lisp
(progn
  (start-profiling :processes :all)
  (sleep 6)
  (stop-profiling))
```
The arguments initialize, processes, profile-waiting and ignore-in-foreign are passed to \texttt{start-profiling}. They have the same default values as for \texttt{start-profiling}, except processes which defaults to \texttt{:all}.

The arguments print and stream are passed to \texttt{stop-profiling}. They have the same default values as in \texttt{stop-profiling}.

\texttt{sleep} is the time to sleep in seconds. If \texttt{sleep} is \texttt{nil} or 0 \texttt{do-profiling} does not sleep. Also, if \texttt{sleep} is supplied and either \texttt{function} or \texttt{func-and-args} are passed, it does not sleep.

\texttt{func-and-args}, and \texttt{function} together with \texttt{arguments}, can both be used for calling a function you supply. \texttt{func-and-args} is either a list of the form \texttt{(function-designator . args)}, in which case \texttt{function-designator} is applied to the \texttt{args}, or it is a function designator which is called without arguments. \texttt{function} is applied to \texttt{arguments}.

The order of execution is first \texttt{func-and-args} (if this is non-nil), then \texttt{function} together with \texttt{arguments} if \texttt{function} is non-nil, and then sleep if \texttt{sleep} was passed explicitly or both \texttt{function} and \texttt{func-and-args} are \texttt{nil}.

\textbf{Example}

To profile whatever happens in the next 6 seconds:

\begin{verbatim}
(hcl:do-profiling)
\end{verbatim}

To profile whatever happens in the next 10 minutes:

\begin{verbatim}
(hcl:do-profiling :sleep 600)
\end{verbatim}

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

**dump-form**  

*Function*

**Summary**  
Dumps selected forms to a stream.

**Package**  
hcl

**Signature**  
dump-form form stream => nil

**Arguments**  
form  
Form to be dumped.

stream  
Stream form is to be dumped to.

**Values**  
Returns nil.

**Description**  
dump-form is used in conjunction with with-output-to-fasl-file to dump selected forms. A dumped form is evaluated when loaded using load-data-file.

See with-output-to-fasl-file for more details.
See also  
dump-forms-to-file
with-output-to-fasl-file

### dump-forms-to-file

**Function**

**Summary**
Dumps specified forms to a fasl file.

**Package**
hcl

**Signature**
dump-forms-to-file pathname forms => nil

**Arguments**
- **pathname**  Name of the fasl file to be created.
- **forms**  Forms to be dumped.

**Values**
Returns nil.

**Description**
dump-forms-to-file dumps specified forms to a fasl file. Use the Common Lisp functions make-load-form and make-load-form-saving-slots to control the dumping of forms.

The best way to specify the file type of the output file is to use compile-file-pathname as in the example below. The file types currently used by LispWorks for fasl files are listed in compile-file.

If the file pathname already exists, it is superseded.

A fasl file created using dump-forms-to-file must be loaded only by load-data-file, and not by load.

**Example**
```
(defclass my-class () 
((a :initarg :a :accessor my-a)))

(defmethod make-load-form ((self my-class) &optional 
environment)
  (declare (ignore environment))
  "(make-instance ',(class-name (class-of self))
   :a ',(my-a self))"

(setq *my-instance* (make-instance 'my-class :a 42))
```
(dump-forms-to-file
  (compile-file-pathname "my-instance")
  (list `(setq *my-instance* ,*my-instance*) ))

In another session, with the same definition of my-class, loading the file "my-instance" using load-data-file will
create an equivalent instance of my-class:

(sys:load-data-file
  (compile-file-pathname "my-instance") )

See also
with-output-to-fasl-file

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
geneneration 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
geneneration 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.

Note: 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

enlarge-static

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.
Note: 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

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.

Note: 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

### 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 note**
In LispWorks 4.4 and previous on Windows and Linux platforms, `extend-current-stack` is not implemented. This is fixed in LispWorks 5.0 and later.

**Example**
To double the size of the current stack:

```
(hcl:extend-current-stack 100)
```

**See also**
- current-stack-length
- *stack-overflow-behaviour*
**extended-time**

**Macro**

**Summary**
Prints useful timing information, including information on garbage collection (GC) activity.

**Package**
hcl

**Signature**
`extended-time &body body`

**Arguments**
`body` The forms to be timed.

**Description**
The macro `extended-time` runs the forms in `body`. It then prints a summary of the time taken followed by a breakdown of time spent in the GC.

The three columns of the GC breakdown show, respectively, total time, user time, and system time. 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 \texttt{marking-gc}) in which the highest generation collected was \( \text{gen-num} \).

Thus in the example below

\texttt{Standard 1 (6 calls) ...}

indicates that there were 6 Standard GCs in which the highest generation collected was 1.

\section*{Example}

This example illustrates output in 32-bit LispWorks:

```
CL-USER 2 > (extended-time (foo))
Timing the evaluation of (FOO)

User time = 7.203
System time = 0.046
Elapsed time = 7.265
Allocation = 84011236 bytes
0 Page faults
Calls to %EVAL = 23000075
```

This example illustrates output in 64-bit LispWorks:

```
```
CL-USER 2 > (extended-time (foo))
Timing the evaluation of (FOO)

User time    =        4.468
System time  =        0.208
Elapsed time =        4.716
Allocation   = 96030696 bytes
0 Page faults

See also

time

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, designat-
ing a file.

length The number of characters to return in string, or nil (the default).

external-format An external format specification, default value :default.

Values string A string containing characters from file.

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**
Checks if 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

**find-object-size**

*Function*

**Summary**
Returns the size in bytes of the representation of any Lisp object.

**Package**
hcl

**Signature**
find-object-size object => size
Arguments

object Any Common Lisp form.

Values

The result is an integer which is the number of bytes of heap 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 string of 10 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

USER 37 > (hcl:find-object-size (make-string 1000 :initial-element #'\A))
1012

See also

room
total-allocation

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 \texttt{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 \texttt{finish-heavy-allocation} is after loading something.

See also \texttt{with-heavy-allocation}

\section*{flag-not-special-free-action}

\textit{Function}

\textbf{Summary}

Unflags an object for special action on garbage collection.

\textbf{Package}

\texttt{hcl}

\textbf{Signature}

\texttt{flag-not-special-free-action object => nil}

\textbf{Arguments}

\textit{object} The object on which the special actions are to be removed.

\textbf{Values}

Returns \texttt{nil}.

\textbf{Example}

\begin{verbatim}
CL-USER 29 : 1 > (make-instance 'capi:title-pane)
#<CAPI:TITLE-PANE "" 20F9898C>

CL-USER 30 : 1 > (flag-not-special-free-action *)
NIL
\end{verbatim}

See also \texttt{add-special-free-action}

\texttt{flag-special-free-action}

\texttt{remove-special-free-action}
flag-special-free-action

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.

Example
CL-USER 29 > (make-instance 'capi:title-pane)
#<CAPI:TITLE-PANE "" 20F9898C>
CL-USER 30 > (flag-special-free-action *)
T

See also
add-special-free-action
flag-not-special-free-action
remove-special-free-action

gc-generation

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.
The function \texttt{gc-generation} does a Garbage Collection of a specific generation. The actual operation is different between 64-bit LispWorks and 32-bit LispWorks.

\textit{gen-num} should be a valid generation number, or \texttt{t}. The value \texttt{t} is mapped to the blocking generation number in 64-bit LispWorks, and to 2 in 32-bit LispWorks. For backwards compatibility the keyword \texttt{:blocking-gen-num} is also accepted, with the same meaning as \texttt{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 \texttt{gc-generation} is when objects are marked for special free action (by \texttt{flag-special-free-action}). If such objects live long enough to be promoted to higher generation, they may not be GCed 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 \textit{gen-num} \texttt{t}).

\textbf{Operation in 64-bit LispWorks}

By default \texttt{gc-generation} operates on the live objects in generation \textit{gen-num} and all lower generations at or above the generation specified by \texttt{block} by copying them inside their current generation, and it operates on the live objects in
generations lower than \textit{block} by copying them to the next higher generation.

If \textit{promote} is non-nil, the live objects in generation \textit{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 \textit{promote} is \texttt{nil}.

If \textit{coalesce} is non-nil, all non-static live objects in lower generations are promoted to generation \textit{gen-num}. That is what \texttt{clean-down} does (with \textit{gen-num} being the highest generation). It may be useful directly in some cases. The default value of \textit{coalesce} is \texttt{nil}.

\textit{block} specifies a generation number up to which to promote. An integer value specifies the generation number. If \textit{block} is \texttt{:blocking-gen-num}, then \texttt{gc-generation} promotes up to the blocking generation. If \textit{block} is \texttt{:all}, then \texttt{gc-generation} promotes nothing. The default value of \textit{block} is \texttt{:blocking-gen-num}.

\texttt{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 \texttt{gc-generation} of generation 2 or generation 3. If you know a time when the application can spend time GCing, a call to \texttt{gc-generation} with a higher value of \textit{gen-num} may be useful. It is probably never really useful to use \texttt{gc-generation} on generation 0 or 1.

To decide on which \textit{gen-num} to call \texttt{gc-generation}, check which generation gets full by making periodic calls to \texttt{room}.

\texttt{gc-generation} with \textit{promote} or \textit{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:

\[
\text{(gc-generation 4 :coalesce t)}
\]

to move all (non-static) objects to generation 4, where they will not be touched by the GC any more (except following pointers to younger generations).

**Operation in 32-bit LispWorks**

`gc-generation` marks and sweeps the generation `gen-num` and all generations below, and then does some additional cleanups. `coalesce`, `promote` and `block` are ignored.

**Compatibility note**

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

\[
\text{(gc-generation t)}
\]

is intended as the replacement of

\[
\text{(mark-and-sweep 2)}
\]

**See also**

- `clean-down`
- `mark-and-sweep`
- `marking-gc`
- `set-blocking-gen-num`
**gc-if-needed**

**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.

**Note:** This function 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

---

**get-default-generation**

**Summary**
Returns the current default generation.

**Package**
hcl
### get-default-generation

**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.

**Note:** 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*

---

### 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.

Note: 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 . 1000)
 (:MAXIMUM-OVERFLOW . 1000000)
 (:MINIMUM-OVERFLOW . 500000)
 (:MINIMUM-BUFFER-SIZE . 200)
 (:NEW-GENERATION-SIZE . 262144)
 (:PROMOTE-MAX-BUFFER . 1000000)
 (:PROMOTE-MIN-BUFFER . 200)
 (:MAXIMUM-BUFFER-SIZE . 131072)
 (:MINIMUM-FOR-SWEEP . 8000)
 (:BIG-OBJECT . 131072))

See also set-gc-parameters

get-temp-directory

Function

Summary

Returns a directory that can be used for temporary files.

Package

hcl

Signature

get-temp-directory => directory

Values

directory A pathname
The function `get-temp-directory` returns a directory which is likely to be writable and can be used for temporary files.

See also `example-compile-file`

**get-working-directory**

*Function*

**Summary**

Finds the current working directory.

**Package**

`hcl`

**Signature**

`get-working-directory => cwd`

**Arguments**

None.

**Values**

`cwd` The current working directory, as a pathname.

**Description**

This function is used to find the current working directory. It returns a pathname, the directory component of which is the current working directory.

**Example**

```
CL-USER 1 > (get-working-directory)
#P"/u/dubya/"
```

See also `cd` `change-directory`

**Variable**

**handle-existing-defpackage**

**Summary**

Controls LispWorks’ response when `defpackage` is used on an existing package that is different from the definition given.

**Package**

`hcl`
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 signalled.

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 signalled 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`, `:shadowing-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`*  

**`*handle-old-in-package-used-as-make-package*` Variable**

Summary Controls the handling of CLtL1-style `in-package` forms.
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*

---

**Variable**

*load-fasl-or-lisp-file*

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

mark-and-sweep

Function
Summary
Garbage collects a specified generation in 32-bit LispWorks.

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 func-
tion 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.

**Note:** `mark-and-sweep` 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 `gc-generation` or `marking-gc`.

**Examples**

```lisp
(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-if-needed`
- `normal-gc`
- `set-array-weak`
- `set-gc-parameters`
- `set-hash-table-weak`
- `without-interrupts`
- `with-heavy-allocation`
**Variable**

*max-trace-indent*  

**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
Function

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*

---

**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.

**Note:** *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*
*packages-for-warn-on-redefinition*  

**Variable**

<table>
<thead>
<tr>
<th>Summary</th>
<th>List of packages whose symbols should be checked for definitions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>hcl</td>
</tr>
<tr>
<td>Initial Value</td>
<td>A list containing &quot;COMMON-LISP&quot; and other package names.</td>
</tr>
</tbody>
</table>
| Description | LispWorks detects attempts to define external symbols in the packages on the list *packages-for-warn-on-redefinition*.

LispWorks, as distributed, is configured to protect the COMMON-LISP package and other system packages.

In particular, the effect of including "COMMON-LISP" in the list value of *packages-for-warn-on-redefinition* is to make all COMMON-LISP symbols be reserved words in respect of definitions and bindings. LispWorks is configured like this because ANSI Common Lisp states that the consequences of such definitions and bindings are undefined. Therefore they are best avoided.

The action taken by LispWorks on such attempted definitions depends on the value of *handle-warn-on-redefinition*.

**See also**  

*handle-warn-on-redefinition*

---

**parse-float**  

**Function**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Parses a float from a string and returns it as float.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>hcl</td>
</tr>
<tr>
<td>Signature</td>
<td>parse-float string &amp;key start end default-format =&gt; float</td>
</tr>
</tbody>
</table>
32 The HCL Package

Arguments

- **string**: A string
- **start, end**: Bounding index designators for string
- **default-format**: One of the atomic type specifiers short-float, single-float, double-float, or long-float.

Values

- **float**: A float

Description

The function `parse-float` parses a float from the substring of string delimited by start and end and returns it as float.

If the substring represents an integer or the exponent marker is E or is omitted, then float will be of type default-format, which defaults to the value of `*read-default-float-format*`. 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 signalled.

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`

**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*.

Example

First set up the profiler:

```lisp
CL-USER 1 > (set-up-profiler
                   :symbols
                   '(cadr car eql fixnump + 1+ caadr cddr))
```

```lisp
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>(%)</td>
<td>top</td>
</tr>
<tr>
<td>CADR</td>
<td>5000012</td>
<td>13</td>
</tr>
<tr>
<td>CDDR</td>
<td>3000000</td>
<td>3</td>
</tr>
<tr>
<td>EQL</td>
<td>2000202</td>
<td>4</td>
</tr>
<tr>
<td>FIXNUMP</td>
<td>2000003</td>
<td>2</td>
</tr>
<tr>
<td>CAR</td>
<td>1000000</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>1000000</td>
<td>3</td>
</tr>
<tr>
<td>CAADR</td>
<td>1000000</td>
<td>2</td>
</tr>
<tr>
<td>1+</td>
<td>1000000</td>
<td>2</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*
### profile  

**Macro**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Runs the specified forms, and prints a performance profile.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>hcl</td>
</tr>
<tr>
<td>Signature</td>
<td>profile &amp;body forms =&gt; final</td>
</tr>
<tr>
<td>Arguments</td>
<td>forms The forms making up the program being profiled.</td>
</tr>
<tr>
<td>Values</td>
<td>final The result of evaluating the final form.</td>
</tr>
<tr>
<td>Description</td>
<td>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 <em>profile-symbol-list</em>. 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.</td>
</tr>
</tbody>
</table>
Examples

USER 22 > (set-up-profiler
   :symbols '(* gethash typep maphash))
NIL
USER 23 > (profile (let ((x 1))
   (loop for a from 1 to 50 by 1
       do (setq x (* a x))
       finally (return x))))
profile-stacks called 12 times
Symbol called profile (%) top
(%)  
MAPHASH  1  0  (0)
0 (0)*  50  1  (8)
0 (0)  
SYSTEM::DUMMY-STRUCTURE-ACCESSOR  6  0  (0)
0 (0)  
SYSTEM::DUMMY-STRUCTURE-SETTER  9  0  (0)
0 (0)  
TYPEP  19  1  (8)
0 (0)  
GETHASH  78  3  (25)
3 (25)  
Top of stack not monitored 75% of the time
3041409320171337804361260816606476884437764156896051200
0000000000

See also

print-profile-list
*profile-symbol-list*
set-up-profiler

*profiler-threshold*  
Variable

Summary
Controls which symbols are profiled on repeated profiling runs.

Package  hcl

Description
*profiler-threshold* is used with repeated profiling runs, to control which symbols are profiled. It is set by set-profiler-threshold.
See also \texttt{set-profiler-threshold}

\texttt{*profile-symbol-list*}

\textit{Variable}

Summary  The list of symbols to be profiled.

Package  \texttt{hcl}

Description  \texttt{*profile-symbol-list*} is the list of symbols that are profiled if \texttt{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  \texttt{nil}

Notes  \texttt{*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  \texttt{add-symbol-profiler}
\texttt{remove-symbol-profiler}
\texttt{set-up-profiler}

\texttt{profiler-tree-from-function}

\textit{Function}

Summary  Prints a call tree of profiled code below a given function.

Package  \texttt{hcl}

Signature  \texttt{profiler-tree-from-function function-name &optional max-depth}

Arguments  \texttt{function-name}  A symbol naming a function.
\texttt{max-depth}  A number or \texttt{nil}. 
The function `profiler-tree-from-function` prints a tree with root `function-name` whose children are the callees of `function-name` and their callees.

`profiler-tree-from-function` uses the data from the previous 'profile session' with style :tree. A profile session ends at the end of `profile` or when `stop-profiling` is called, or when the Profiler tool finishes profiling.

In both cases the counts of profile calls is the total counts of the calls to `function-name`. Note that the percentages (the number in parentheses) are percentages from the total number of profile calls, rather than from the numbers of calls to `function-name`.

If `max-depth` is a number it limits the depth of tree that is printed to that value. The default value of `max-depth` is `nil`, meaning no limit on the depth that is printed.

See also

- `profile`
- `start-profiling`
- `stop-profiling`

`profiler-tree-to-function` function

<table>
<thead>
<tr>
<th>Summary</th>
<th>Prints a reversed call tree of profiled code below a given function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td><code>hcl</code></td>
</tr>
<tr>
<td>Signature</td>
<td><code>profiler-tree-to-function function-name &amp;optional max-depth</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>function-name</code> A symbol naming a function.</td>
</tr>
<tr>
<td></td>
<td><code>max-depth</code> A number or <code>nil</code>.</td>
</tr>
<tr>
<td>Description</td>
<td>The function <code>profiler-tree-to-function</code> prints a tree with root <code>function-name</code> whose children are the callers of <code>function-name</code>.</td>
</tr>
</tbody>
</table>
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`

**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`.
Note: The cross-referencing information used by references-who is generated when code is compiled with source-level debugging switched on.

See also toggle-source-debugging
who-references

remove-special-free-action

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

Summary
Removes a symbol from the list of profiled symbols.

Package hcl
remove-symbol-profiler  

**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

save-argument-real-p

Function

Summary

Used to determine if a build script knows the real name of the image being saved.

Package

hcl

Signature

save-argument-real-p => realp

Arguments

None

Values

realp A boolean.

Description

The function save-argument-real-p can be used in a build script to determine if the argument passed to a subsequent call to save-image or deliver is the real filename of the application.

The return value realp is t in most cases. It is nil only when building an intermediate image for the purpose of building a universal binary, either by save-universal-from-script or the Application Builder (see the LispWorks IDE User Guide).
Operations in a build script that are related to the path of the saved image, such as building an application bundle, should be executed only when this function returns \texttt{t}. When using \texttt{save-universal-from-script}, any required application bundle should be created before calling that function (see the \texttt{save-macos-application.lisp} example below). When using the Application Builder, any required application bundle should be created in the build script only when \texttt{save-argument-real-p} returns \texttt{t}.

On architectures that do not have universal binaries, this function always returns \texttt{t}.

\begin{description}
\item[Example] examples/configuration/save-macos-application.lisp
\item[See also] save-universal-from-script, building-universal-intermediate-p, deliver, save-image, save-image-with-bundle
\end{description}

### save-current-session

\textit{Function}

\begin{description}
\item[Summary] Saves the LispWorks session.
\item[Package] \texttt{hcl}
\item[Signature] \texttt{save-current-session pathname &rest save-image-args => result}
\item[Arguments] pathname \hspace{1em} A pathname designator.
\hspace{1em} save-image-args \hspace{1em} Arguments.
\item[Values] result \hspace{1em} A boolean.
\item[Description] The function \texttt{save-current-session} closes all windows and stops multiprocessing, saves an image at the location sup-
plied in pathname, and restarts multiprocessing and the windows. For more information see “Saved sessions” on page 133.

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. Whilst save-current-session can be used to save a session in a console image, this achieves nothing more than save-image.

2. The released LispWorks image runs the default session. Therefore after you have used save-current-session, starting the supplied image (for example via the Windows start menu or MacOS X Dock) will run itself only if the default session is "LispWorks Release".

See also
save-image
save-image-with-bundle

save-image

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  

filename  A string. It is the name of the file that the image is saved as. This name should not be the same as the original name of the image.

dll-exports  A list of strings, or the keyword :default.

dll-added-files  A list of strings.

automatic-init  A generalized boolean.

gc  If non-nil, there is a garbage collection before the image is saved. The default value is t.

type  Determines if some global variables are cleared before the image is saved. You can generally use the default value, which is :user.

normal-gc  If this is t the function normal-gc is called before the image is saved. The default is t.

restart-function  A function to be called on restart.

multiprocessing  Controls whether multiprocessing is enabled on restart.

console  On Windows console controls whether the new image will be a Console or GUI application and when, if ever, to make a console window in the latter case.

On the Macintosh console controls when, if ever, to make a console window.

Possible values are discussed below.

environment  environment controls whether the LispWorks environment is started on restart. Possible values are discussed below.

remarks  remarks adds a comment to the save history. The value should be a string.

clean-down  When t, calls (clean-down t).
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 12, “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 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

### image-type
One of `:exe`, `:dll` or `: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 12, “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 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 13, “LispWorks as a dynamic library”.

dll-added-files should be a list of filenames. It is ignored on Microsoft Windows. On other platforms if dll-added-files is non-nil then a dynamic library containing each named file is saved. Each file must be of a format that the default C compiler (scm:*c-default-compiler*) knows about and can incorporate into a shared library. Typically they will be C source files, but can also be assembler or object files. They must not contain exports that clash with names in the LispWorks shared library (see Chapter 45, “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 306.
- Calling QuitLispWorks when required.
- Changing calls that involve complex C structs or even C++ objects into plain calls, because accessing C structures in Lisp requires defining the structure, while in C it only needs to include the header.
- Creating 'stub' functions that can be called from Lisp, for example for calling a C++ method. The address of the stub function can be passed to Lisp which can call it using a function defined by fli:define-foreign-funcallable.
- Adding code that runs automatically inside the call to dlopen, by using __attribute__((constructor))

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. restart-function is called after the initialization file is loaded. The default value of restart-function is nil.

Note: restart-function is not called if the command line argument -no-restart-function is present.
When `multiprocessing` is `nil`, the executable image will start without multiprocessing enabled. When `multiprocessing` is true or the image is a DLL, the image will start with multiprocessing enabled. The default value of `multiprocessing` is `nil`.

`console` is implemented only in LispWorks for Windows and LispWorks for Macintosh. The possible values for `console` are as follows:

- `:default` Unchanged since previous save.
- `t` On the Macintosh, the value `t` has the same effect as the value `:always`.
  On Windows, a Console application is saved, else a Windows application is saved which creates its own console according to the other possible values.

- `:input`, `:output`, `:io`
  Whenever input, output or any I/O is attempted on `*terminal-io*`.

- `:init` At startup, if input and output are not redirected.

- `:always` At startup, even if input and output are redirected.

The LispWorks for Windows and LispWorks for Macintosh images shipped have `console` set to `:input`.

The possible values for `environment` are as follows:

- `:default` Unchanged since previous save.
- `nil` Start with just the TTY listener.
- `t` Start the environment automatically, no TTY listener.
:with-tty-listener

Start the environment automatically, but still have a TTY listener.

The LispWorks image shipped is saved with :environment t on all platforms except for the Motif images on Mac OS X, Solaris, HP-UX and DEC Tru64 UNIX.

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

Do not supply :multiprocessing nil along with a true value of :environment t. Multiprocessing is needed for the GUI environment.

Compatibility note

LispWorks 5.0 and previous versions documented -init as the way to run LispWorks with a build script. This method is deprecated.

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.

In LispWorks 5.0 and previous versions dll-exports is supported only on Windows.

dll-added-files and automatic-init are new in LispWorks 5.1.

Example

Here is an example build script. Save this to a file such as c:/build-my-image.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/macos-application-bundle")
 :load t)
(save-image
 #:cocoa
 (write-macos-application-bundle
  "/Applications/LispWorks 6.0/My LispWorks.app"
  #:cocoa
  "my-lispworks")
```

See also
- `deliver`
- `dll-quit`
- `InitLispWorks`
- `LispWorksDlsym`
- `load-all-patches`
- `quit`
- `QuitLispWorks`
- `save-current-session`

### 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**
- hci

**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` checks `save-argument-real-p`, so it can be used for saving universals without further checks.

`save-image-with-bundle` operates as follows:

1. If `save-argument-real-p` returns true, it calls `bundle-function` with the `bundle-path` and `bundle-arguments`, and then uses the result as the filename for `save-image`. Otherwise, the filename for `save-image` is `nil`.

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`).

`save-image-with-bundle` is implemented only in LispWorks for Macintosh.

See also

- `create-macos-application-bundle`
- `save-image`
<table>
<thead>
<tr>
<th>save-universal-from-script</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
<td>Saves a universal binary LispWorks image using a script designed for saving a mono-architecture image.</td>
</tr>
<tr>
<td><strong>Package</strong></td>
<td>hcl</td>
</tr>
<tr>
<td><strong>Signature</strong></td>
<td><code>save-universal-from-script target-image script-name &amp;key output-stream =&gt; target-image</code></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td></td>
</tr>
<tr>
<td>target-image</td>
<td>A pathname designator.</td>
</tr>
<tr>
<td>script-name</td>
<td>A pathname designator.</td>
</tr>
<tr>
<td>output-stream</td>
<td>A stream or nil.</td>
</tr>
<tr>
<td><strong>Values</strong></td>
<td>target-image A pathname designator.</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The function <code>save-universal-from-script</code> provides a convenient way to create a universal binary on an Intel Macintosh, using a script designed for saving a mono-architecture image. The <code>script-name</code> is the name of a Common Lisp build script for saving or delivering an image, as would be used to create a mono-architecture image. It should load the application and then call either <code>deliver</code> or <code>save-image</code> as appropriate. The function <code>save-universal-from-script</code> runs the current LispWorks image in two subprocesses, once for the PowerPC architecture (under Rosetta) and once for the native Intel architecture, passing <code>-build script-name</code> on the command line. The script is evaluated as normal, except that the filename that is passed to any call to <code>save-image</code> or <code>deliver</code> is ignored and a temporary filename is used instead. If these two subprocesses are successful, then the temporary images are combined to make a universal binary <code>target-name</code> in the same way as <code>create-universal-binary</code>.</td>
</tr>
</tbody>
</table>
The command line arguments of the images run by the subprocesses will include the command line arguments that were passed to the current image. In addition, various undocumented command line arguments will be prepended, which control how `deliver` or `save-image` work in the script.

Any output generated by the subprocesses is written to `output-stream`. If this is `nil`, then the output is discarded. If this is `t` (the default), then the output is written to the standard output.

**Note:** The function `save-universal-from-script` can only be called from a LispWorks for Macintosh image that is itself a universal binary, such as the distributed image.

### Example
Suppose the file `my-build-script.lisp` contains

```lisp
(load-all-patches)
(load "my-application-defsys")
(compile-system 'my-application-system :load t)
(deliver 'my-application-function "my-application" 5)
```

Then, the following call creates a universal binary `my-application` using this script:

```lisp
(save-universal-from-script "my-application"
   "my-build-script.lisp")
```

### See also
- `save-image`
- `create-universal-binary`
- `building-universal-intermediate-p`
- `save-argument-real-p`

### set-array-single-thread-p

**Function**

**Summary**
Tells the system whether an array is accessed only in a single thread context, or not.

**Package**
`hcl`
Signature  

set-array-single-thread-p array on-p

Arguments  

array  An array.
on-p  A generalized boolean.

Description  

Tells the system whether the array array is accessed only in a single thread context or not, depending on the value of on-p. Arrays that are marked for single thread access are faster for some operations, in particular vector-push and vector-pop.

See also  

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 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.

Note: 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

---

**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).

Note: set-default-generation is useful only in 32-bit Lisp-
Works. 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*

set-gc-parameters

Function

Summary

Sets the parameters from the garbage collector in 32-bit Lisp-
Works.

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

maximum-buffer-size

Maximum size of the small objects buffer.

minimum-buffer-size
Minimum size of the small objects buffer.

**big-object**

An object that is bigger than this value is “big”. That is, it is not allocated from the small objects buffer, but from the big-chunk area (if it is allocated in generation 0 in the normal way).

**promote-min-buffer**

During promotion, a buffer is allocated in the generation being promoted into, and the objects promoted are moved into it. *promote-min-buffer* controls the minimum size of this buffer.

**promote-max-buffer**

Controls the maximum size of the promotion buffer.

**new-generation-size**

Controls the minimum enlargement of generation *gen-num*, for *gen-num > 0*. Value 0 means the generation is not expanded. Otherwise, *new-generation-size* must be a fixnum in the exclusive range (10000, 100000000) and the minimum expansion is then *new-generation-size* × *gen-num* words. *new-generation-size* has no effect on the enlargement of generation 0.

**maximum-overflow**

Maximum size of the small-objects buffer in the big-chunk area.

**minimum-overflow**

Minimum size of the small-objects buffer in the big-chunk area.

**minimum-for-promote**
Controls the frequency of promotions. Setting `minimum-for-promote` to a high value causes the system to promote less frequently. This may improve performance for programs that allocate a lot of data for a short term and then delete it.

`minimum-for-sweep`

Controls when a mark-and-sweep takes place. Setting `minimum-for-sweep` to a high value causes the system to mark and sweep less often, which means it has to grow. The CPU time spent in garbage collection is mostly smaller, but the process is bigger and may cause more disk access.

`enlarge-by-segments`

A minimum for how much the image grows each time a segment is enlarged, as a multiple of 64K. 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.

**Note:** `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`
32 The HCL Package

Package  hcl

Signature  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 point-
                        ers 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 argu-
                            ments.

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 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 make-hash-table
mark-and-sweep
set-array-weak

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 generation-size A list showing information for the generation just specified in the call.

Description Sets the minimum free space for a segment of the specified generation.
By default, affects the first segment — pass `segment` to affect a different segment of the generation.

The minimum free space is shown by `room`.

**Note:** `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`

---

### 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`.

The `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.

The `processes` controls which processes are added to the set, removed from the set, or 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
set-promotion-count is deprecated, because experience has shown that it is not useful.
set-promotion-count is implemented only in 32-bit Lisp-Works. It is not relevant to the Memory Management API in 64-bit implementations, wherein you may be able to achieve the effect with set-delay-promotion.

See also

block-promotion
clean-generation-0
collect-generation-2
collect-highest-generation
expand-generation-1

---

**set-system-message-log**

*Function*

**Summary**

Manipulates the system message log.

**Package**

hcl

**Signature**

```lisp
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 \textit{stream} is \texttt{t} or a stream, the system message log stream is set, with \texttt{t} meaning \texttt{*standard-output*}. This stream is used when writing messages.

When \textit{collect} is true but not :\texttt{no-change}, messages are collected in an internal list, which can be retrieved by using \textit{get}.

\textit{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 \textit{stream}, \textit{collect} and \textit{callback} is :\texttt{no-change}, which does not change the current setting.

When \textit{get} is supplied \texttt{set-system-message-log} returns a list of the messages that has been collected. Each message is a single string. If \textit{get} is \texttt{t}, the internal list is reset to \texttt{nil}. If \textit{get} is :\texttt{keep}, the internal list is not reset, so the next call with \textit{get} will get them again.

\texttt{set-system-message-log} returns \texttt{nil} if \textit{get} is not supplied.

\texttt{set-system-message-log} returns the list of collected messages if \textit{get} is supplied.

\begin{itemize}
  \item \texttt{stream}, \texttt{callback} and \texttt{collect} are mutually independent. It is possible to set the system to any combination of these.
  \item The order of operation when a message is generated is first to print, then call the callback, and then collect.
  \item When collecting messages it can accumulate, so it is important to periodically get the message to ensure it does not bloat the memory.
  \item Using \texttt{collect \texttt{t}} when it is already collecting has no effect, in particular it does not affect the list of collected messages.
\end{itemize}

\textbf{set-up-profiler}  \\
\textbf{Function}  \\
\textbf{Summary}  \\
\texttt{ 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
- `symbols` A symbol or a list of symbols.
- `packages` A valid package name, or a list of package names, or :all.
- `kind` :profile, :virtual or :real.
- `interval` An integer greater than or equal to 10000.
- `limit` An integer or nil.
- `cutoff` An integer or nil.
- `collapse` A generalized boolean.
- `style` :tree, :list or nil.
- `gc` A generalized boolean.
- `call-counter` A generalized boolean.
- `show-unknown-frames` A generalized boolean.

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.
The profiler will generate a tree of calls seen in the profiler, as well as the output shown by :list.

The default value of style is :tree.

gc specifies whether to profile functions inside the memory management code (more accurately, functions that are called on the GC stack) in addition to any other profiling. The default value of gc is nil.

call-counter 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).

Note: 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.

Note: call-counter is effective only on x86 platforms or in 64-bit LispWorks. On non-x86 platforms 32-bit LispWorks decides whether to do call counting for each function when it is compiled, depending on the debug level, and call-counter has no effect.

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.

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*
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.

Note: The cross-referencing information used by sets-who is
generated when code is compiled with source-level debug-
ning switched on.

See also
who-sets
toggle-source-debugging

source-debugging-on-p  

Function

Summary  Tests if source level debugging is on for compiled code.
The HCL Package

Package: hcl

Signature: source-debugging-on-p => bool

Arguments: None.

Values: bool

If t, source level debugging is on.

Description: Returns t if source level debugging is on for compiled code; otherwise returns nil.

See also: toggle-source-debugging

start-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 \texttt{set-process-profiling}.

\texttt{profile-waiting} is used only in SMP LispWorks. When \texttt{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.

\texttt{ignore-in-foreign} controls whether to ignore processes that are inside foreign calls. The default value of \texttt{ignore-in-foreign} is \texttt{nil}.

\texttt{start-profiling} can be repeatedly called without intervening calls to \texttt{stop-profiling}, for example to change the setting of \texttt{profile-waiting} or the profiled processes.

\texttt{start-profiling} cannot be used while \texttt{profile} is used or while the Profiler tool is profiling (on any thread). Between the call to \texttt{start-profiling} and the next call to \texttt{stop-profiling} with \texttt{print t} (or omitted), \texttt{profile} and the Profiler tool cannot be used.

Various parameters which are set by \texttt{set-up-profiler} control the behavior of the profiler. See the documentation for \texttt{set-up-profiler}.

\textbf{Examples}

The following sequence of calls to \texttt{start-profiling} and \texttt{stop-profiling} can be used to profile only interesting work and print the results:

Start profiling the current process:

\begin{verbatim}
  (start-profiling :processes :current)
  (do-interesting-work)
\end{verbatim}

Temporarily suspend profiling:

\begin{verbatim}
  (stop-profiling :print nil)
  (do-uninteresting-work)
\end{verbatim}

Resume profiling:
(start-profiling :initialize nil)
(do-more-interesting-work)
(stop-profiling)

See also
profile
do-profiling
set-process-profiling
stop-profiling

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.

Description
The function stop-profiling stops collecting profiling information, and optionally prints the results.

If print is non-nil, the information collected so far is printed and the next call to start-profiling must pass initialize t or omit the initialize argument. If print is nil, then the profiler is put into a suspended state where no profiling information is collected, but can be restarted by calling

(start-profiling :initialize nil)

The default value of print is t.

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*.

Note: parameters set by set-up-profiler control the format of the output.
sweep-all-objects

Function

Summary
Applies a function to all the live objects in the image.

Package
hcl

Signature
sweep-all-objects function &optional gen-0 => nil

Arguments
function
A function of one argument, the object.

gen-0
A generalized boolean, default value nil

Values
sweep-all-objects returns nil.

Description
Applies function to all the live objects in the image. Normally it is not useful to sweep objects in generation 0 because they are ephemeral, so by default sweep-all-objects does not sweep generation 0. This can be changed by passing a non-nil value as gen-0.

function should take one argument, the object. It can allocate, but if it allocates heavily the sweeping becomes unreliable. Small amounts of allocation will normally happen only in generation 0, and so will not affect sweeping of other generations.

To call sweep-all-objects reliably, do it inside with-other-threads-disabled.

Notes
In 64-bit LispWorks 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
do-profiling
profile
set-process-profiling
start-profiling
The HCL Package

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**
switch-static-allocation returns the previous setting of flag.

**Description**
Objects in the static area are garbage-collected, but not moved.

You should avoid using this function.

See also enlarge-static
in-static-area

**symbol-alloc-gen-num**

*Variable*

**Summary**
Specifies the generation in which interned symbols and their symbol names are allocated.

**Package**
hcl

**Initial Value**
2 in 32-bit LispWorks, 3 in 64-bit LispWorks
See also  
allocation-in-gen-num
get-default-generation
set-default-generation

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.

*notice-changed-definitions*  
When true, the Cross Referencer notices when a function is redefined, including an interpreted redefinition..

*source-level-debugging*  
When true, the compiler generates information used by the debugger.

toggle-source-debugging modifies the status of the variables, and then returns the new value. To check whether all the variables are set to true, without modifying them, use source-debugging-on-p.

Cross-referencing information is used by the functions who-calls, who-binds, who-references, who-sets, and friends.

Compatibility note  
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**  

*Summary* Calculate memory consumed since the image was started.

*Package* hcl

*Signature* total-allocation
Arguments: None.

Values: Returns the amount allocated.

Description: This function calculates the total amount of memory consumed since the current image was created. Use at the start and end of a piece of code, to see how much it allocates.

See also: find-object-size, room

---

*traced-arglist*  

**Summary:** The list of arguments given to the function being traced.

**Package:** hcl

**Initial Value:** nil

**Description:** Upon entering a function that is being traced, *traced-arglist* is bound to the list of arguments given to the function. *traced-arglist* is then printed after the function name in the output from tracing. It is accessible in the :before and :after forms to trace. However care should be used when manipulating this variable, since it is the value of *traced-arglist* itself that is used when calling the traced function. Thus if this value is altered by the :before forms then the function receives the altered argument list.

**Example:**

```
USER 14 > (trace (+ :before
   ((setq *traced-arglist*
     (mapcar #'1+
     *traced-arglist*)))))

+USER 15 > (+ 1 2 3)
```
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 
awei ((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

\*trace-indent-width\*  

Variable

Summary  
The amount of extra indentation in the trace output for each level of nesting.

Package  
hcl

Initial Value  
2

Description  

*traced-results* is the extra amount by which the traced output for function calls is indented upon entering a deeper level of nesting (i.e. a traced call from a function that is itself traced). If it is 0 then no indentation occurs.
32 The HCL Package

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 * < ...:
   >> 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.

See also
trace

*trace-level*  Variable

Summary
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
Example

USER 8 > (defun fac (n) (if (<= n 1)
            1
            (* n (fac (1- n)))))

FAC

USER 9 > (trace fac)

FAC

USER 10 > (fac 3)

0 FAC > (3)
  1 FAC > (2)
  2 FAC > (1)
  2 FAC < (1)
  1 FAC < (2)
0 FAC < (6)
  6

Notes  *trace-level* is an extension to Common Lisp.

See also  trace

**trace-print-circle**

Variable

Summary  Controls how circular structure are printed in trace output.

Package  hcl

Initial Value  nil

Description  *trace-print-circle* controls how circular structures are
printed during output from tracing. It allows the printing of
Circular structures by the tracer to be controlled independ-
ently 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)
        l)

CIRC

USER 21 > (trace second)
SECOND

USER 22 > (second (circ '(1 2 3 4)))
0 SECOND > (#1=(1 2 3 4 . #1#))
0 SECOND < (2) 2

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.

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)
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 subcomponents are at level 2, and so on.

*print-level* is bound to the value of *trace-print-level* while printing tracing information. If *trace-print-level* is nil then objects are printed without regard to depth.

Examples USER 8 > (trace append)
APPEND
USER 9 > (dotimes (i 10) (setq li (if (zerop i)
   nil
   (list i li))))
NIL
USER 10 > (append li '(a b))
0 APPEND > ((9 (8 (7 (6 #)))) (A B))
0 APPEND < ((9 (8 (7 (6 #))) A B))
(9 (8 (7 (6 (5 (4 (3 (2 (1 NIL)))))))) A B)

Notes *trace-print-level* is an extension to Common Lisp.

See also trace

*trace-print-pretty* Variable

Summary Controls the amount of whitespace in trace output.

Package hcl

Initial Value nil

Description *trace-print-pretty* controls the amount of whitespace printed during output from tracing. If it is not nil then extra whitespace is inserted to make the output more comprehensible. *print-pretty* is bound to the value of *trace-print-pretty* while printing tracing information.

Examples USER 6 > (trace macroexpand-1)
MACROEXPAND-1
USER 7 > (setq *trace-print-pretty* t
   *print-pretty* nil)
NIL
USER 8 > (defmacro sum (n)
   '(do ((i 0 (1+ i))
   (res 0 (+ i res)))
   ((= i ,n) res)))
```
SUM
USER 9 > (macroexpand-1 'sum 3))
0 MACROEXPAND-1 > (SUM 3)
0 MACROEXPAND-1 < (DO ((I 0 (1+ I))
(RES 0 (+ I RES)))
(= I 3)
RES))
T)
(DO ((I 0 (1+ I)) (RES 0 (+ I RES))) (= I 3) RES))
T
```

Notes
*trace-print-pretty* is an extension to Common Lisp.

See also
trace

*trace-verbose*

Variable

Summary
Controls how arguments and values are printed in trace output.

Package
hcl

Initial Value
:only

Description
*trace-verbose* controls the way arguments and values are printed in trace output.

If the value is not nil then trace attempts to decode the arguments and values, and prints them.

When the value is :only, trace does not print the lists of arguments and values after the function name.

Notes
*trace-verbose* is an extension to Common Lisp.

See also
trace
**try-compact-in-generation**

**Function**

**Summary**
Comacts the most fragmented segment(s) in a generation in 32-bit LispWorks.

**Package**
hcl

**Signature**
```
try-compact-in-generation generation-number time-threshold
&optional fraction-threshold => result
```

**Arguments**
- `generation-number`
  0 for the most recent generation, 1 for the most recent two generations, and so on up to a maximum (usually 3). Numbers outside this range signal an error.

- `time-threshold`
  A real number.

- `fraction-threshold`
  A real number between 0 and 1, defining the minimum fragmentation to actually compact. The default is 0.25.

**Values**
- `result`
  A boolean.

**Description**
`try-compact-in-generation` finds the most fragmented segment in the generation specified. If `time-threshold` is positive, it compacts this segment, and repeats this operation until `time-threshold` seconds have elapsed. At this point `try-compact-in-generation` returns, with value `t` if at least one segment was compacted and value `nil` otherwise. Because the operation cannot be stopped in the middle, the actual time taken will always be larger than `time-threshold`.

If `fraction-threshold` is 1, `try-compact-in-generation` does nothing. If `fraction-threshold` is 0, `try-compact-in-generation` will compact all uncompacted segments (unless it runs out of time). With the default (0.25)
try-compact-in-generation compacts only moderately fragmented segments.

If time-threshold is negative, then try-compact-in-generation does not actually compact any segments. result is a boolean indicating whether try-compact-in-generation would actually try to compact a segment if it were to be called with a positive time-threshold and the other arguments unchanged.

This function is typically used after a call to check-fragmentation. For more information, see “Controlling Fragmentation” on page 110.

Note: 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

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.
The HCL Package

**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.

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>A boolean.</td>
</tr>
</tbody>
</table>

**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) try-move-in-generation moves only moderately fragmented segments.

If time-threshold is negative, then try-move-in-generation does not actually move any segments. result is a boolean indicating whether try-move-in-generation would actually try to move a segment if it were to be called with a positive time-threshold and the other arguments unchanged.

This function is typically used after a call to check-fragmentation. For more information, see “Controlling Fragmentation” on page 110.

**Note:** try-move-in-generation is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations, where marking-gc with the what-to-copy argument offers similar functionality (although
set-blocking-gen-num is intended to solve the problem of fragmentation automatically).

See also  
check-fragmentation  
try-compact-in-generation

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

who-binds

*Function*

Summary
Returns the definitions which bind a special variable.

Package
hcl

Signature
who-binds symbol => result

Arguments
symbol A special variable.

Values
result A list.
The function `who-binds` returns a list of dspecs naming the definitions which bind the special variable `symbol`.

**Note:** The cross-referencing information used by `who-binds` is generated when code is compiled with source-level debugging switched on.

See also `binds-who`
`toggle-source-debugging`

---

**who-calls**

**Function**

**Summary**

Returns the callers of a function.

**Package**

`hcl`

**Signature**

`who-calls dspec => callers`

**Arguments**

`dspec` A dspec.

**Values**

`callers` A list.

**Description**

The function `who-calls` returns a list of dspecs naming the definitions which call the function named by `dspec`.

See also the Editor commands List Callers and Show Paths To.

**Note:** The cross-referencing information used by `who-calls` is generated when code is compiled with source-level debugging switched on.

See also `calls-who`
`toggle-source-debugging`
**who-references**  
*Function*

**Summary**
Returns the definitions which reference a special variable.

**Package**
hcl

**Signature**
who-references symbol => result

**Arguments**
symbol  
A special variable.

**Values**
result  
A list.

**Description**
The function **who-references** returns a list of dspecs naming the definitions which reference the special variable *symbol*.

**Note:** 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-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*.
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`

### with-hash-table-locked

**Macro**

**Summary**
Evaluates code with a hash-table locked against modification by other threads.

**Package**
hcl

**Signature**

```
with-hash-table-locked hash-table &body body => results
```

**Arguments**

- `hash-table` A hash table.
- `body` Forms.

**Values**

- `results` The results of evaluating `body`.

**Description**

The macro `with-hash-table-locked` evaluates `body` with the hash table `hash-table` locked against modification by other threads. The current thread can modify `hash-table`.

`with-hash-table-locked` is useful not only for multiple accesses to the same table, but also when an access to the table must be consistent with some other operation, avoiding the need to make a separate lock.

See also

- `make-hash-table`
- `modify-hash`
**with-heavy-allocation**  
*Macro*

**Summary**  
Slows up garbage collection during the execution of code that allocates a lot of space.

**Package**  
hcl

**Signature**  
(with-heavy-allocation &rest body => result)

**Arguments**  
body  
The forms for which you want the garbage collector to behave differently from normal.

**Values**  
result  
The result of executing body.

**Description**  
The macro with-heavy-allocation is for use with code that allocates a lot of space but is not interactive. It ensures that garbage collection (GC) is carried out less frequently while these forms are being executed. However, each GC may take longer.

**Compatibility note**  
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

**with-output-to-fasl-file**  
*Function*

**Summary**  
Sends output to a fasl file on disk.
Package  hcl

Signature  with-output-to-fasl-file (stream pathname &rest options)
&body body => nil

Arguments  stream  Stream to be bound to the fasl file to be created.
pathname  Name of the fasl file to be created.
body  Forms, some of which may be dumped.

Values  Returns nil.

Description  with-output-to-fasl-file is used in conjunction with dump-form. The body forms are executed, and during the execution, dump-form may be called to dump selected forms. Dumped forms are evaluated if the file pathname is later loaded by load-data-file.

Supply an appropriate fasl extension in pathname. A simple way to achieve this is by calling compile-file-pathname. A complete list of fasl extensions for supported platforms may be found in compile-file.

If the file pathname already exists, it is superseded.

A fasl file created using with-output-to-fasl-file must be loaded only by load-data-file, and not by load.

Example  CL-USER 12 > (with-output-to-fasl-file (s "/tmp/foo.fasl")
(dump-form '(print 'hello) s))
NIL

CL-USER 13 > (let ((sys:*binary-file-type* "fasl"))
(sys:load-data-file "/tmp/foo.fasl")
; Loading fasl file "/tmp/foo.fasl"

HELLO
#P="/tmp/foo.fasl"
See also
dump-form
dump-forms-to-file
load-data-file
The LINK-LOAD 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, or x86/x64 Solaris).

### break-on-unresolved-functions

**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 break-on-unresolved-functions function 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 `foreign-symbol-address` function 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 signalled. If it is `nil`, no error will be reported, and the function will return `nil`.

The `functionp` keyword is used to specify the kind of symbol sought. If it is `t`, `foreign-symbol-address` will assume that
name is the name of a function. If it is nil it will assume that name is the name of a variable. The default value is t.

Example

(foreign-symbol-address 'chmod)

See also

get-foreign-symbol

get-foreign-symbol

Function

Package link-load

Signature get-foreign-symbol name &optional force => result

Arguments

name A symbol or string.

force A keyword.

Values

result A foreign symbol.

Description

This function gets a foreign symbol or it may be used to explicitly register an undefined symbol.

name is a symbol or string to look up or to create as a foreign symbol. If it is a symbol, the symbol looked for is that which the function lisp-name-to-foreign-name would produce. If name is a string, it is taken literally.

If supplied and the symbol is not already defined as a foreign symbol, force forces it to be an undefined foreign symbol. This provides a reference to the symbol so that a subsequent call to read-foreign-modules will attempt to resolve it.

Example

(get-foreign-symbol 'my-func-not-yet-loaded t)

Notes

It is not usually necessary to use this function. In order to examine whether a foreign symbol is defined, use foreign-symbol-address. The act of defining a foreign function using
fli:define-foreign-function makes the symbol undefined, so the use of force is not usually needed.

See also
- foreign-symbol-address
- lisp-name-to-foreign-name
- read-foreign-modules

### lisp-name-to-foreign-name

Function

**Package**  
link-load

**Signature**  
lisp-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
This chapter applies only to LispWorks for UNIX

**Function**

**read-foreign-modules**

**Package**

link-load

**Signature**

`read-foreign-modules &rest module-names => t`

**Arguments**

`module-names` A sequence of strings or pathnames.

**Values**

`t`

**Description**

The `read-foreign-modules` function 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**

```lisp
(read-foreign-modules "/usr/users/clc/projects/head.o" 
"~/clc/projects/libs.a" 
"-lw")
```

**Notes**

The `read-foreign-modules` function actually adds the module-names to the list of modules in the variable *default-libraries* and then tries to resolve any undefined symbols using this list. The function `get-foreign-symbol` may be
called to explicitly force a symbol onto the undefined list or
the act of defining a foreign function (\texttt{fli:define-foreign-}
\texttt{function}) will do it implicitly.

\texttt{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 \texttt{*unresolved-}
\texttt{messages*} is set to \texttt{t} (the default is \texttt{nil}). By default messages
are printed out about which object modules are being loaded.
This may be switched off by setting \texttt{*coff-loading-}
\texttt{verbose*} to \texttt{nil}.

\textbf{See also} \texttt{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

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

**Type**

**Summary**
The 16 bit string type.

**Package**
lispworks

**Signature**
16-bit-string length

**Arguments**
length
The length of the string (or *, meaning any).

**Description**
The type of strings that can hold simple chars of codes 0…65533. This is the string type that is guaranteed to always take 16 bits per element.

### appendf

**Macro**

**Summary**
Appends lists to the end of a given list.

**Package**
lispworks

**Signature**
appendf place &rest lists => result

**Arguments**
place
An object.

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

### base-character

**Type**

**Summary**
The base character type.
Package   lispworks
Signature  base-character
Description The type of base characters.

   base-character is a synonym for the Common Lisp type base-char.

See also   base-char-code-limit

base-character-p  Function
Summary   Tests if an object is a base character

Arguments  object   The object to be tested.

Values    bool     t if object is a base character; nil otherwise.

Description This is the predicate for base characters.

See also   base-character

base-char-p  Function
Summary   Tests if an object is a base character

Arguments  object   The object to be tested.
Values: bool, t if object is a base character; nil otherwise.

Description: This is also the predicate for base characters, only with standard spelling.

See also: base-character-p

**base-char-code-limit**

Summary: Upper bound for character codes in base characters.

Package: lispworks

Description: The upper exclusive bound for values of (char-code char) among base characters.

**base-string-p**

Summary: Tests if an object is a base string.

Package: lispworks

Signature: base-string-p object => bool

Arguments: object, The object to be tested.

Values: bool, t if object is a base string; nil otherwise.

Description: This is the predicate for base strings.

See also: base-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 require-
ment 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 original 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 `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 `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 `call-next-advice`. Around advice may contain any number of calls to `call-next-advice`, including no calls.

**Notes**

`call-next-advice` is an extension to Common Lisp. See Chapter 6, “The Advice Facility” for a broader discussion of advice.

**See also**

`defadvice`
choose-unicode-string-hash-function  Function

Summary  Returns a hash function suitable for strings, ignoring case using specified Unicode rules.

Package  lispworks

Signature  choose-unicode-string-hash-function &key style => hash-function

Arguments  style  A keyword

Values  hash-function  A hash function

Description  The function choose-unicode-string-hash-function return a function which is suitable for use as the hash-function argument to make-hash-table. The function hash-function generates a hash value for a string, ignoring case using specified Unicode comparison rules specified by style.

The current implementation only supports one value of style:

:simple-case-fold  Compares each character of the string using Unicode's simple case folding rules.

See also  make-hash-table
           unicode-string-equal

compile-system  Function

Summary  The function compile-system compiles all the files in a system necessary to make a consistent set of object files.

Package  lispworks

Signature  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 `defsystem` macro.

**force**  
If `t` then all the files in the system are compiled regardless. (This argument was formerly called `force-p`. The old name is currently still accepted for compatibility.)

**simulate**  
If `nil` or not present then `compile-system` works silently. Otherwise a plan of the actions which `compile-system` intends to carry out is printed. What happens next depends on the value of `simulate`:
- `t` — do nothing.
- `:ask` — you are asked if you wish the plan to be carried out using `y-or-n-p`.
- `:each` — `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 `c` executes the rest of the plan without further prompting, returns from `compile-system` without further processing, and `y` and `n` work as expected.

`:simulate` may be abbreviated as `:sim`.

**load**  
If `t` then `load-system` is called after `compile-system` has finished. If `:no` then no files are loaded at all. The default is `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.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**
scm

**Signature**
concatenate-system output system &key force simulate sim
source-only args target-directory t-dir script-p => result

  system ::= system-name*
Arguments

**output** The name of the required concatenated fasl.

**system-name** The name of a system defined using `defsystem`.

**simulate** Verbosity conditions, see Description for more detail.

**sim** Same as `simulate`.

**force** If `t`, then all files in the system will be concatenated.

**source-only** If `t`, the source files of the system are concatenated.

**target-directory** The directory to search for the object files.

**t-dir** Same as `target-directory`.

Values

**result** A list containing the name or names of the concatenated systems.

Description

This function produces a single, concatenated fasl, `output-file`, from a list of individual systems (named amongst the `args`).

Since concatenated fasl files may be produced in this way, you do not need to be wary of MS filename conventions if developing sources on UNIX for a Microsoft Windows application. This clearly allows more freedom for naming source files. However, `output-file` must, in such cases, be a MS-Win-
dow-compatible filename.

If `simulate` is `nil` or is not present, `concatenate-system` will work silently. Otherwise, a plan of the actions which `concatenate-system` intends to carry out is printed. What happens next depends upon the value of `simulate`:

- If it is `t`, the function does nothing.
- If `:ask`, then the user is asked, using `y-or-n-p`, if the plan should be carried out.
• If it is :each, the user is asked at each stage in the plan if the current action should be carried out. The responses y and n work as normal. If e is typed, \texttt{concatenate-system} exits without further processing.

If \texttt{source-only} is \texttt{t}, files will be loaded only if they are sources.

If, when searching \texttt{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.

\textbf{See also} \texttt{compile-system} \hfill \texttt{defsystem} \hfill \texttt{load-system}

**current-pathname** *Function*

\textbf{Summary} Computes a pathname relative to the current path.

\textbf{Package} \texttt{lispworks}

\textbf{Signature} \texttt{current-pathname &optional relative-pathname type => pathname}

\textbf{Arguments} \textit{relative-pathname} A pathname designator. \hfill \textit{type} A string or \texttt{nil}. The default is \texttt{nil}.

\textbf{Values} \textit{pathname} A pathname.

\textbf{Description} The function \texttt{current-pathname} is useful for loading other files relative to a file.

\texttt{current-pathname} computes a pathname from the current operation as follows:

When loading a file

\begin{verbatim}
Uses *load-pathname*.
\end{verbatim}
When compiling a file
  Uses \texttt{*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 \texttt{relative-pathname} is merged with this physical pathname. The \texttt{pathname-type} of the result \texttt{pathname} is set to \texttt{type} if supplied, the \texttt{pathname-version} is set to \texttt{:newest}, and \texttt{pathname} is returned.

A useful value for \texttt{type} is \texttt{nil}, which can be used to allow \texttt{load} to choose between lisp or fasl regardless of the type of the current pathname.

\textbf{Note:} \texttt{defsystem} uses \texttt{current-pathname} with its \texttt{:default-host} argument.

**Examples**

Suppose you want the file \texttt{foo} to load the file \texttt{bar}.

While loading the source file \texttt{foo.lisp}:

\begin{verbatim}
(c current-pathname "bar")
=> #P"C:/temp/bar.lisp"
\end{verbatim}

While loading the binary file \texttt{foo.ofasl}:

\begin{verbatim}
(c current-pathname "bar")
=> #P"C:/temp/bar.ofasl"
\end{verbatim}

To load \texttt{bar.lisp} or \texttt{bar.ofasl} according to the value of \texttt{*load-fasl-or-lisp-file*}, regardless of whether \texttt{foo.lisp} or \texttt{foo.ofasl} is being loaded, specify \texttt{type nil}:
(load (current-pathname "bar" nil))

See also defsystem pathname-location

defadvice

Macro

Summary Defines a new piece of advice.

Package lispworks

Signature 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  
Contains 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`

**`*default-character-element-type*`**  
*Parameter*  

Summary  
Provides defaults for all character type parameters.

Package  
lispworks

Description  
This variable provides defaults for all character type parameters. The legal values are `base-char`, `lw:simple-char`, and
character. Its value must only be set via a call to `lw:set-default-character-element-type`.

This is intended mainly for running old 8-bit applications efficiently. If you write for a fat character implementation 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 `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 `define-action` macro adds a new action to the specified list; this action will be executed according to the action-list’s execution-function (see `execute-actions`) when executed. If the action-list specified by `name-or-list` does not exist, then
this is handled according to the value of *handle-missing-action-list*.

`name-or-list` is evaluated to give either a list UID (to be looked up in the global registry of lists) or an action list object. `action-name` is a UID (general lisp object, to be compared by `equalp`). It uniquely identifies this action within its list (as opposed to among all lists).

`data` specifies an object referring to data relevant to the action.

`specs` is a free-form list of ordering specifiers and extra keywords, used to control more details of how and when this action is executed.

Action-items are normally expected not to be redefined. If an action-item with that action-name already exists in the action-list (that is, one with an identifier `equalp` to the action-name), then the notification and subsequent handling of this attempt is controlled by the values in the list `*handle-existing-action-in-action-list*`. This is to prevent problems due to re-evaluating an action definition inappropriately.

Notification and redefine behavior can be overridden by using the `:force` keyword argument. In this case, any required redefinition is performed unconditionally and without notification.

The following keywords are recognized in the `specs` argument:

`:after` The following element in `specs` is a UID. `

`:after` specifies that the action-item being defined must be run after the action-item named. If there is no action-item with a matching name, the restriction is ignored.

`:before` Like `:after`, but this action-item must be run before the one specified.

`:after` and `:before` can be specified as many times as necessary to describe the ordering constraints of this action-item with respect to its neighbors.
:once Specifies that this action-item should be executed only once; after execution, it is disabled.

:force Specifies that this definition should override any previous definition of this action-item, rather than be subject to the value of *handle-existing-action-in-action-list*.

Example

(define-action :network-startup "Reset decnet buffers"
  '(decnet::reset-network-buffers
    *net-buffers*)
  :after "Reset core network"
  :once))

See also

undefined-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.

Description

The define-action-list macro 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 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 `*handle-existing-action-list*` variable.

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-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 also
  *default-action-list-sort-time*
  *handle-existing-action-list*
  undefine-action-list

**defsystem**

*Macro*

**Summary**
defsystem is used to define systems for use with the Lisp-Works 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” on page 193.

**Package**
lispworks

**Signature**
defsystem system-name options &key members rules => system

**Arguments**
system-name The name of the system to be made.

options are expressed as a list of keyword argument pairs. The following keywords are recognized:

:package The default package that files are compiled and loaded in. If not specified, this defaults to the value of *package* at macroexpansion time.

:default-pathname Used to compute a default pathname in which to find files. defsystem uses current-pathname to compute the pathname. defsystem checks that all the files given as members actually exist.
:default-host
The root pathname of a system is defined to be the :default-host if it is given. Otherwise, it is taken to be the directory containing the defsystem file.

Absolute pathnames are interpreted literally, and relative pathnames are taken relative to the root pathname.

:default-type
This is the default type of the members of the system. This may be :lisp-file, :lsp-file, :c-file, or :system.


The com module adds the type :midl-file and the automation module adds :midl-type-library-file.

The default is :lisp-file, which means files with file type (extension) "lisp".

:documentation
This is a string.

:object-pathname
A string or pathname specifying a directory where object files are written.

Note: This option will not work if the names in members represent absolute pathnames.

:optimize
A declaration specifying default compilation qualities within the scope of compile-system. These settings override the current global setting. They can be overridden per member by the :optimize option (for subsystems) or proclaim (in files). The :optimize defsystem option accepts the
same optimize qualities as proclaim and which are fully described in “Compiler control” on page 88. See below for examples.

members is a list defining the members of the system. Each element of the list may be a symbol or a string representing the name of the physical file or system referred to, or a list of format (name {keyword value}+) where name is once again a symbol or a string referring to the system or physical file, and the possible keywords 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.

On Windows, the automation module adds the keyword :com for a member with type :midl-type-library-file. Then a member of the form

("msoc97.tlb" :type :midl-type-library-file :com nil)
can be specified when you use only Automation client code, reducing the memory used.

Rules is a list of rules of the following format:

\[
\{\text{:in-order-to} \text{ action} \{\text{:all} \mid \{\text{ member-name } \}^* \}\} \\
\{\text{:caused-by} \{\text{ action} \{\text{:previous} \mid \{\text{ member-name } \}^* \}\}^*\} \\
\{\text{:requires} \{\text{ action} \{\text{:previous} \mid \{\text{ member-name } \}^* \}\}^*\}\}
\]

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.

There are more details about the rules in “DEFSYSTEM rules” on page 196.

Values

The name of the system is returned.

Examples

(defsystem defsys-macros
  (:default-pathname "/usr/users/james/scm/defsys/
  :default-type :lisp-file
  :package defsystem)
  :members ("new-macros"
            "scm-timemacros"))
(defsystem clos-sys
  (:default-pathname "/usr/users/clc/defsys/"
  :default-type :lsp-file
  :package defsystem)
 :members
  "class"
  "time-methods"
  "scm-pathname" :source-only t)
 :rules

(defun system-list)

(defun compile-system

(defun load-system

(defun member-system

(defun rule-system

(defun defsystem dataworks-demo
  (: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 last example illustrates the use of :optimize.

(defsystem foo (:optimize ((speed 3) (space 3)
  (safety 0)))
 :members (*bar"
  "baz")
 :rules ((:compile :all
    (:requires (:load :previous))))
Notes

Systems that are members of another system must be declared in the system declaration file before the system of which they are a part.

The ordering of members is important and reflects the order in which operations are carried out on the members of the system.

See also

load-system
compile-system
concatenate-system
current-pathname
*defsystem-verbose*

*defsystem-verbose*  

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  

Summary

Deletes a directory.

Package

lispworks
Signature  
**delete-directory**  **directory**  &optional  **error**  =>  **result**

Arguments  
**directory**  A pathname designator.
**error**  nil, :error or :no-error.

Value  
**result**  t or nil.

Description  
The function **delete-directory** attempts to delete the directory **directory**. It returns t on success, and on failure either returns nil or signals an error.

**error** determines what happens when **delete-directory** fails. When **error** is nil (the default), if **directory** does not exist **delete-directory** returns nil, otherwise any failure causes an error to be signaled. If **error** is :no-error, **delete-directory** returns nil on any failure. If **error** is :error, any failure causes an error to be signaled.

Typical reasons for failures in **delete-directory** are that **directory** is not empty, or that the user does not have the right permissions.

---

**deliver**  
*Function*

Summary  
The main interface to the Delivery tools.

Package  
lispworks

Signature  
**deliver**  **function**  **file**  **level**  &rest  **keywords**

Description  
The function **deliver** is the main interface to the LispWorks delivery tools. You use it to create LispWorks executable applications and dynamic libraries.

For more information about Delivery including a detailed description of **deliver**, see the *LispWorks Delivery User Guide*.  

---

695
For information about invoking \texttt{deliver} using the IDE, see "The Application Builder" in the \textit{LispWorks IDE User Guide}.

See also \texttt{save-image}

\textbf{*describe-length*} \hfill \textit{Variable}

<table>
<thead>
<tr>
<th>Summary</th>
<th>Determines how many attributes of a composite object are described.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>\texttt{lispworks}</td>
</tr>
<tr>
<td>Initial Value</td>
<td>20</td>
</tr>
<tr>
<td>Description</td>
<td>The variable \texttt{<em>describe-length</em>} controls how many attributes of a composite object the function \texttt{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 \texttt{<em>describe-length</em>} is \texttt{nil} then \texttt{describe} describes all of the attributes. Use this value only with care. Note: the \texttt{describe} functionality is load-on-demand in the LispWorks image as shipped. Therefore if you have not done (\texttt{require &quot;<em>describe</em>&quot;}) or called \texttt{describe}, \texttt{<em>describe-length</em>} may be unbound.</td>
</tr>
</tbody>
</table>

See also \texttt{describe}

\textbf{*describe-level*} \hfill \textit{Variable}

| Summary               | Controls the depth to which \texttt{describe} describes arrays, structures and conses. |
Package lispworks

Initial Value 1

Description The variable *describe-level* controls the depth to which the function describe describes arrays, structures and conses.

Note: the describe functionality is load-on-demand in the LispWorks image as shipped. Therefore if you have not do (require "describe") or called describe, *describe-level* may be unbound.

Example

CL-USER 23 > (describe 1)
[... load output not shown ...]

1 is a BIT
DECIMAL 1
HEX 1
OCTAL 1
BINARY 1

CL-USER 24 > *describe-level*
1

CL-USER 25 > (defstruct foo a s d)
FOO

CL-USER 26 > (defmethod describe-object ((f foo) (s stream))
    (format s "FOO ~S~S"
    (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 `:dll-exports` or by `deliver` with `:dll-exports`, and
- Inside the dynamic scope of a callback into the dynamic library. That is, not in a process that was started by `process-run-function`.

`dll-quit` sets up the internal state such that just before returning into its caller in the LispWorks dynamic library it causes LispWorks to quit. After quitting the callback returns as normal. The library can be unloaded using `FreeLibrary`, or you can re-use it (without re-loading).
By default `kill-all-processes` is `nil` which means that, if there are other running processes, `dll-quit` just returns `nil`. If `kill-all-processes` is non-nil, `dll-quit` tries to kill all the other processes, and if it succeeds, it quits.

If `kill-all-processes` is true, `timeout` is a maximum time to wait after killing the other processes. It allows `timeout` seconds for all processes to die.

`dll-quit` should be called when no other processes are running, whether they were created by a callback or by `process-run-function`. If such processes exist, by default `dll-quit` does nothing and returns `nil`. If `force` is non-nil, `dll-quit` always tries to set LispWorks up for quitting. LispWorks will quit even after a failure to kill all other processes and complete any required shut down operations. A true value of `force` automatically implies `kill-all-processes` true. However, if any of the other processes is stuck in a foreign call, the quitting may fail to finish properly. The default value of `force` is `nil`.

If `output` is supplied, `dll-quit` generates output if it is called when other processes are still running, or a required shut down operation was not completed. `output` can be an output stream, `t` (interpreted as `*standard-output*) or `nil`. If `output` is `nil`, `dll-quit` collects the output and returns it as second argument `quit-output`. Otherwise it writes the output to the stream and `quit-output` is `nil`.

The output contains a list of the other processes that are still running. If `kill-all-processes` or `force` was supplied, and killing the other processes failed, the output also contains backtraces of the other processes, and possibly other debugging information.

`result` is `t` on success: the LispWorks dynamic library is set to quit on returning from the callback. `result` is `nil` when other processes are running: the image is not set to quit.
quit-output contains the output which was generated when output nil was passed. Otherwise quit-output is nil.

If dll-quit is called inside a recursive foreign callback, the LispWorks dynamic library quits only when the outermost callback returns.

**Note:** 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 Freelibary to free the library. The shutdown call should close and free everything that needs to be closed or freed, call dll-quit, and return.

**Note:** 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.
The function `dotted-list-length` performs the same action as `list-length`, except that if the last `cdr` is not `nil` then instead of signalling 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 => bool`

**Arguments**
- `list` A list, which must be a `cons`.

**Values**
- `bool` 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 a true value if this is the case, otherwise it returns nil.

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

| Values | unspecified | An unspecified value.

### Description

The function **do-nothing** ignores its arguments and returns an unspecified value. It is useful as a function argument.

### See also

false
true

### *enter-debugger-directly*

**Variable**

| Summary | Controls direct entry into the Debugger tool.
| Package | lispworks
| Initial value | nil

### Description

The variable **enter-debugger-directly** is a generalised boolean which affects the behavior of the LispWorks IDE when an error is signalled 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.

**Note:** Errors signalled in a Listener Read-Eval-Print loop are handled in the REPL and therefore **enter-debugger-directly** has no effect on the behavior in this case.

### environment-variable

**Function**

| Summary | Reads the value of an environment variable from the environment table of the calling process.
Package        lispworks

Signature      environment-variable name => result

Arguments      name          A string.

Values         result        A string or nil.

Description    The function environment-variable reads the environment variable specified by name and returns its value, or nil if the variable could not be found.

A setf method is also defined, allowing you to set the value of an environment variable:

(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:\user\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\MOUS E"

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")
```

### `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`.

**Note:** this 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
description of arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>A pathname designator.</td>
</tr>
</tbody>
</table>

Values
description of values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>A pathname.</td>
</tr>
</tbody>
</table>

Description
description of function

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/6-0-0-0/examples/capi/applications/othello.lisp"
```

See also
description of related functions

`example-compile-file`
path to an output file in a temporary location which is likely to be writable.

It then calls \texttt{compile-file} with these paths as the \textit{input-file} and \textit{output-file}, also passing the other \texttt{args}, and returns the values returned by \texttt{compile-file}.

See also \texttt{get-temp-directory} \\
\texttt{example-file}

\section*{example-load-binary-file \textit{Function}}

\textbf{Summary} \quad Loads a fasl file compiled by \texttt{example-compile-file}.

\textbf{Package} \quad \texttt{lispworks}

\textbf{Signature} \quad \texttt{example-load-binary-file file => generalized-boolean}

\textbf{Arguments} \quad \textit{file} \quad A pathname designator.

\textbf{Values} \quad \texttt{generalized-boolean} \quad The value returned by \texttt{load}.

\textbf{Description} \quad The function \texttt{example-load-binary-file} constructs the path to an output file in a temporary location which would be used as the \textit{output-file} by \texttt{example-compile-file}.

It then calls \texttt{load} on that path, and returns the values returned by \texttt{load}.

See also \texttt{example-compile-file}

\section*{execute-actions \textit{Macro}}

\textbf{Summary} \quad Executes in sequence the actions on a given list.
The LISPWORKS Package

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.

Description

The execute-actions macro 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 signalled 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**
Tests if an object is an extended character.

**Package**
*lispworks*

**Signature**
`extended-character-p object => bool`

**Arguments**
`object` The object to be tested.

**Values**
`bool` `t` if `object` is an extended character; `nil` otherwise.

**Description**
This is the predicate for extended characters.

**See also**
`extended-character`

---

**extended-char-p**  
*Function*

**Summary**
Tests if an object is an extended character.

**Package**
lispworks

**Signature**
`extended-char-p object => bool`

**Arguments**
`object` The object to be tested.

**Values**
`bool` `t` if `object` is an extended character; `nil` otherwise.

**Description**
This is also the predicate for extended characters, only with standard spelling.

**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:

```lisp
```

On all other platforms:

```lisp
```

**Description**  
The variable *external-formats* contains a list of the names of the defined external formats.

The platform-specific external format names are:

- **code-page**  
  Uses the encoding in the Microsoft Windows code page specified by the :id parameter.

- **latin-portable**  
  Intended for use when communicating with X servers, for example when passing XLFD names. Uses the X Portable Character Set.
host-portable

A synonym for latin-portable.

false

Function

Summary
Ignores its arguments and returns nil.

Package
lispworks

Signature
false &rest ignore -> nil

Arguments
ignore
All arguments are ignored.

Value
nil

Description
The function false takes any number of arguments, which it ignores, and returns nil. It is useful as a functional argument.

See also
do-nothing
ttrue

file-directory-p

Function

Summary
Tests for the presence of a directory.

Package
lispworks

Signature
file-directory-p pathname => bool

Arguments
pathname
A pathname, string, or file-stream.

Values
bool
If t, the pathname represented by pathname exists and is a directory. If nil, it either does not exist, or it is not a directory.
Description  
file-directory-p tests whether the pathname represents a directory.

Example  
CL-USER 70 > (file-directory-p "--")
T

CL-USER 71 > (file-directory-p ".login")
NIL

**find-regexp-in-string**  
*Function*

Summary  
Matches a regular expression.

Package  
lispworks

Signature  
find-regexp-in-string pattern string &key start end from-end case-sensitive => pos, len

Arguments  
*pattern*  
A string or a precompiled regular expression object.

*string*  
A string.

*start, end*  
Bounding index designators of string.

*from-end*  
A generalized boolean.

*case-sensitive*  
A generalized boolean.

Values  
*pos*  
A non-negative integer or nil.

*len*  
A non-negative integer or nil.

Description  
The function **find-regexp-in-string** searches the string **string** for a match for the regular expression **pattern**. The index in **string** of the start of the first match is returned in **pos**, and the length of the match is **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**

713
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.

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.

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 precompiled regular expression object:
(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 function is the function whose arguments are required
If error-p is nil, then function-lambda-list returns :none if
function is not defined, and does not start the debugger. The
default value of error-p is t, meaning that an error is signalled
if function is undefined.

Example TEST 2 > (function-lambda-list 'editor:create-buffer-command)
(EDITOR::P &OPTIONAL EDITOR:BUFFER-NAME)
get-inspector-values  

**Generic Function**

**Summary**
Customizes the information displayed in the LispWorks IDE Inspector tool.

**Package**
lispworks

**Signature**
get-inspector-values object mode

**Arguments**
- **object**  The object to be inspected.
- **mode**  Name of a mode, or nil. nil defines the default inspection format for object.

**Values**
Returns five values: names, values, getter, setter and type. names and values are the two lists displayed in columns in the inspector window. getter is ignored. setter is a function used to updated slot values. type is displayed at the foot of the inspector window.

**Description**
This generic function allows you to customize the LispWorks IDE Inspector by adding 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.

**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 `get-unix-error` function returns the text associated with the specified value of the UNIX variable `errno`.

Note: this is implemented only on UNIX/Linux/Mac OS X/FreeBSD.

See also `errno-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*`  

### Variable

**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*`
**Variable**

### *grep-fixed-args*

- **Package**: lispworks
- **Summary**: Arguments added to the command string of a Grep search in the Search Files tool.
- **Initial Value**: "-n"
- **Description**: The variable *grep-fixed-args* provides arguments added to a Grep command string in the Search Files tool. The value should ensure that the line number is output at the start of each match.
  
  See the *LispWorks IDE User Guide* for more information about the Search Files tool.

- **See also**: *grep-command*  
  *grep-command-format*

### *handle-existing-action-in-action-list*

- **Summary**: Contains keywords determining behavior on exceptions raised when an action definition already exists in a given action list.
- **Package**: lispworks
- **Initial value**: (:warn :redefine)
- **Description**: 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 define-action-list macro.

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

Description A keyword; one of :warn, :error, or :ignore, denoting how to handle an operation on a missing action-list. The default value is :error. It is used by undefined-action-list, print-actions, execute-actions, define-action and undefine-action.
See also define-action
execute-actions
print-actions
undefine-action
undefine-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 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 *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.

See also

*packages-for-warn-on-redefinition*
*redefinition-action*

**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.
The LISPWORKS Package

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*  
Variable

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.

*inspect-through-gui*  
Variable

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.

---

### lisp-image-name  
*Function*

Summary  
Returns the name of the running image.

Package  
lispworks

Signature  
lisp-image-name => name

Arguments  
None.

Values  
name  
A string.

Description  
The function lisp-image-name returns a string representing the full path to the running LispWorks image. The example below is in typical LispWorks for Windows and LispWorks for Linux installations. In resaved and delivered images (including dynamic libraries such as Windows DLLs), the appropriate path is returned.

Example  
On Windows:

```
CL-USER 1 > (lisp-image-name)
"C:\\Program Files\\LispWorks\\lispworks-6-0-0-x86-win32.exe"
```

On Linux:
CL-USER 1 > (lisp-image-name)
"/usr/bin/lispworks-6-0-0-x86-linux"

See also
*line-arguments-list*

**lispworks-directory*** Variable

Summary
The main LispWorks installation directory.

Package
lispworks

Initial value
The initial value is

#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"C:\Program Files\LispWorks\" on Microsoft Windows.

#P"/Applications/LispWorks 6.0/Library/" on Mac OS X.

Note however that the value can be set when configuring an image or on startup.

Description
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/6-0-0-0/, 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/6-0-0-0/` 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 140 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

**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**

(loadsystem 'blackboard)

(loadsystem '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
- `defsyste\m`
- `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 subsequent-ly defined action-items where no explicit ordering con-
straints are specified. An example is

'(:after :beginning :before :end)

eexecution-function specifies a user-defined function accepting
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, re-
spectively. 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.
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*

*mt-random-state* Variable

Summary The default random state used by mt-random.

Package lispworks

Description The variable *mt-random-state* contains an object of type mt-random-state which is the default state used by mt-random if a state is not supplied.

*mt-random-state* is analogous to cl:*random-state*.

See also make-mt-random-state
mt-random
mt-random-state

mt-random-state Type

Summary The type of objects containing state information used by mt-random.

Package lispworks
Description  The Mersenne Twister pseudo-random number generator uses state data contained in a object of type \texttt{mt-random-state}.
\texttt{mt-random-state} is analogous to \texttt{cl:random-state}.

See also  *\texttt{mt-random-state}*
\texttt{mt-random}
\texttt{mt-random-state-p}

\textbf{mt-random-state-p}  

\textit{Function}

Summary  The predicate for objects of type \texttt{mt-random-state}.

Package  lispworks

Signature  \texttt{mt-random-state-p arg => result}

Arguments  \textit{arg}  
An object.

Values  \textit{result}  
A boolean.

Description  The function \texttt{mt-random-state-p} returns \texttt{t} if \textit{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}  

\textit{Function}

Summary  Returns the location of a file.

Signature  \texttt{pathname-location pathname => location}

Arguments  \textit{pathname}  
A pathname designator.
Values

| location | A pathname. |

Description

The function `pathname-location` returns a pathname `location` that represents the directory where the file `pathname` resides. Each of the name, type and version components of `location` are `nil`.

Example

Due to the ANSI Common Lisp definition of the `directory` function and the fact that LispWorks returns fully specified truenames, the form

```
(directory (truename "/tmp/"))
```

will always signal an error or return the list `#P"/tmp/*"`. To obtain the contents of the `/tmp` directory, use the form

```
(directory (pathname-location (truename "/tmp/")))
```

See also

`current-pathname`
`directory`

---

### precompile-regexp

**Function**

**Summary**

Precompiles a regular expression object.

**Package**

`lispworks`

**Signature**

`precompile-regexp string => 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 `find-regexp-in-string`.

See also

`find-regexp-in-string`
print-actions

Function

Summary
Generates 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 A stream.

Description
Generates a listing of the action items on this action-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*.

stream is an optional argument specifying where to print the output. 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 actions lists in the global registry.

Package
lispworks

Signature
print-action-lists &optional stream

Arguments
stream A stream.

Description
Generates a listing of all the action lists in the global registry. The ordering of the action lists is random.
stream is an optional argument specifying where to print the output. 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 UNIX/Linux/Mac OS X/FreeBSD 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**

- **status**
  An integer.

- **confirm**
  A generalized boolean.

- **ignore-errors-p**
  A generalised boolean.

- **return**
  A generalized boolean.

**Values**

`quit` does not return, or returns `t`.

**Description**
The function `quit` exits LispWorks unless the user cancels the operation.

There are two stages which may allow the user the chance to cancel.

1. First the action items of the action list "Confirm when quitting image" are run. If any action item returns `nil`, then LispWorks does not exit.

2. Otherwise, if `confirm` is true (the default value is `nil`) then a question like
   
   "Do you really want to exit LispWorks?"
   
   is presented to the user. If the answer No is supplied, then LispWorks does not exit. Otherwise, the action items of the action list "When quitting image" are run, and then LispWorks exits, and the value `status` is returned to the Operating System as the exit value of the LispWorks process. The default value of `status` is `0`.

If `ignore-errors-p` is true, then any error signalled during the running of the action list items or the confirm prompt is ignored and `quit` proceeds to exit the image. If `ignore-errors-p`
is `nil` and an error is signalled during the running of the action list items, then a restart is available allowing the user to choose to continue to exit the image. The default values of `ignore-errors-p` is `nil`.

If `return` is true and LispWorks is going to exit, then `quit` returns `t`. This can be used if you want some other Lisp 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 `return` is `nil` then `quit` does not return. The default value of `return` is `nil`.

**Note:** To make a Cocoa application quit cleanly from inside the Quit menu command you need to call `capi:destroy` on the application interface instead of calling `quit`. See `capi:default-cocoa-application-interface` in the CAPI Reference Manual for more information.

**See also**  
`save-image`

---

### rebinding

**Macro**

**Summary**
Ensures unique names for all the variables in a groups of forms.

**Package**
lispworks

**Signature**
`rebinding (&rest vars) &body body => form`

**Arguments**
- `vars` The variables to be rebound.
- `body` A body of forms, the variables in which should be unique.

**Values**
Returns the body wrapped in a form that creates unique names for each variable.
Description

Returns the body wrapped in a form which creates a unique name for each of the variables (compare with gensym) and binds these names to the values of the variables. This ensures that the body can refer to the variables without name clashes with other variables elsewhere.

Example

After defining

```lisp
(defmacro lister (x y)
  (rebinding (x y)
    '(list ,x ,y)))
```

the form `(lister i j)` macroexpands to

```lisp
(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

\[
\text{dspec} ::= \text{fn-name} | \\
\quad \text{macro-name} | \\
\quad \text{(method generic-fn-name} [(\text{class*})])
\]

Arguments  

\(\text{dspec}\)  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.

\(\text{name}\)  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  

remove-advice returns nil.

Description  

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 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).

hcl:delete-advice is a macro, identical in effect to remove-advice, except that you do not need to quote the arguments.
Notes

*remove-advice* is an extension to Common Lisp.

See also

defadvice
defdelete-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 A float

Values double-float A double-float with single-float precision.

Description The argument is rounded to single-precision format (32 bits) and returned as a double-float (64 bits). This function allows you to model the rounding behavior of a machine or implementation that performs 32-bit floating point arithmetic.

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 Note
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

set-default-character-element-type
Function
Summary
Configures the value of lw:*default-character-element-type*.

Package
lispworks
Signature  
set-default-character-element-type  

Arguments  
type  
A character type. This can take any of the values base-char; lw:simple-char and character.

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.

See also  
string
open
*default-character-element-type*
with-output-to-string

simple-base-string-p  

Function

Summary  
Tests if an object is a simple base string.

Package  
lispworks

Signature  
simple-base-string-p object => bool
Arguments

object

The object to be tested.

Values

bool

\( t \) if object is a simple base string; \( \text{nil} \) otherwise.

Description

This is the predicate for simple base strings.

See also

simple-base-string

simple-char

Summary

The simple character type.

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-p

Summary

Tests if an object is a simple character.

Package

lispworks

Signature

simple-char-p object => bool

Arguments

object

The object to be tested.

Values

bool

\( t \) if object is a simple character; \( \text{nil} \) otherwise.

Description

The predicate for simple characters.
simple-text-string

Summary
The simple text string type.

Package
lispworks

Signature
simple-text-string length

Arguments
length The length of the string (or *, meaning any).

Description
This is the simple version of text-string, that is, the string itself is simple. Equivalent to:

(simple-vector lw:simple-char length)

See also
text-string

simple-text-string-p

Summary
Tests if an object is a simple text string.

Package
lispworks

Signature
simple-text-string-p object => bool

Arguments
object The object to be tested.

Values
bool t if object is a simple text string; nil otherwise.

Description
This is the predicate for simple text strings.

See also
simple-text-string
**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**

```lisp
split-sequence-if-not predicate sequence &key start end key coalesce-separators => sequences
```

**Arguments**

- **predicate**  
  A function designator.
- **sequence**  
  A sequence.
- **start, end**  
  Bounding index designators for sequence.
- **key**  
  A function designator or `nil`.
- **coalesce-separators**  
  A generalized boolean.

**Values**

- **result**  
  A list of sequences.

**Description**

The function `split-sequence-if-not` returns a list of subsequences of `sequence` (between start and end), split by where the function `predicate` returns false for an element. The structure of `sequence` is not changed and the elements identified by the predicate are not included in the resulting sequences.

If non-nil, the function `key` is applied to the elements of `sequence` before `predicate` is called.

If `coalesce-separators` is true, then empty sequences are omitted from `result`.

**See also**

- `split-sequence`
- `split-sequence-if`
**start-tty-listener**  

**Function**  

Summary  
Starts a listener in the startup shell.

Package  
lispworks

Signature  
`start-tty-listener force => process`

Arguments  
`force`  
A generalized boolean.

Values  
`process`  
A listener process, or `nil`.

Description  
The function `start-tty-listener` returns a process that runs a listener read-eval-print loop connected to `*terminal-io*`. If `force` is `nil`, then `start-tty-listener` checks if the default listener process is alive or if there is a live process with name "TTY Listener". If such a process exists, `start-tty-listener` simply returns `nil` and does not start a new process. If no such process exists, or if `force` was `t`, then `start-tty-listener` starts a new listener process named "TTY Listener", and returns it.

If a REPL with I/O through `*terminal-io*` (such as a REPL started by `start-tty-listener`) is in the debugger, then by default it blocks multiprocessing. This behavior is controlled by the value of `*terminal-debugger-block-multiprocessing*`.

See also  
`*terminal-debugger-block-multiprocessing*`

**stchar**  

**Function**  

Summary  
The accessor for simple text strings.

Package  
lispworks
**Stchar**

**Signature**

\[ \text{stchar \ string \ index} \rightarrow \text{value} \]

**Arguments**

- **string**: A simple-text-string.
- **index**: An index.

**Values**

- **value**: The character in `string` at `index`.

**Description**

This is the accessor for simple text strings. `setf` is allowed.

**See also**

- simple-text-string

---

**string-append**

**Function**

**Summary**

Constructs a single string from a number of strings.

**Package**

lispworks

**Signature**

\[ \text{string-append \ \&rest \ strings} \rightarrow \text{string} \]

**Arguments**

- **strings**: Any number of strings or string designators.

**Values**

- **string**: A string.

**Description**

The `string-append` function takes any number of string designators and constructs a single string from them.

A string designator is a string, a symbol or a character object.

Each of the elements of the `strings` argument are first coerced into a string using the `string` function if they are 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

```lisp
(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
```

text-string

**Type**

**Summary**
The text string type.

**Package**
lispworks

**Signature**
text-string length

**Arguments**

- **length**
  The length of the string (or *, meaning any).

**Description**
The type of strings that can hold any simple character, that is, (vector lw:simple-char length). This is the string type that is guaranteed to always hold any character used in writing text (program text or natural language). It will not hold character objects which have non-null attributes.

It is equivalent to 16-bit-string.

**See also**

- 8-bit-string
- 16-bit-string
text-string-p

Function

Summary
Tests if an object is a text string.

Package
lispworks

Signature
text-string-p object => bool

Arguments
object
The object to be tested.

Values
bool
t if object is a text string; nil otherwise.

Description
This is the predicate for text strings.

See also
text-string

true

Function

Summary
Ignores its arguments and returns t.

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 `undefine-action` macro removes the specified action from the specified list. If the action specified by `action-name` does not exist, then this is handled according to the value of `*handle-missing-action-in-action-list*`.

`name-or-list` is evaluated to give either a list UID (to be looked up in the global registry of lists) or an action list object. `action-name` is a UID (general lisp object, to be compared by `equalp`). It uniquely identifies this action within its list (as opposed to among all lists).

**See also**
define-action

---

**undefine-action-list**

*Macro*

**Summary**
Removes a given defined action list.

**Package**
lispworks

**Signature**
`undefine-action-list uid =>`

**Arguments**
- `uid` A lisp object.

**Values**
None.
The **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 `*handle-missing-action-list*` variable.

**See also**
- `define-action-list`

---

### `unicode-alpha-char-p`  

**Function**

**Summary**

Returns a value like `cl:alpha-char-p`, but using specified Unicode rules.

**Package**

`lispworks`

**Signature**

`unicode-alpha-char-p char &key style => flag`

**Arguments**

- `char` A character
- `style` A keyword

**Values**

- `flag` A generalized boolean

**Description**

The function `unicode-alpha-char-p` returns `flag` as true if `char` is an alphabetic character according to the Unicode rules specified by `style`.

The current implementation only supports one style:

`:general-category`  

Use Unicode's "general category" for `char`.

**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 Unicode's "general category" for char.

See also
unicode-alpha-char-p
unicode-both-case-p

unicode-both-case-p

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
The LISPWORKS Package

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 Unicode's "general category" for `char`.

Notes

The name of `unicode-both-case-p` is slightly confusing, because it matches the ANSI Common Lisp definition "a character with case" whereas there is no guarantee that both cases actually exist. Note also that there are some "alpha" chars which are not lower or upper case.

See also

`unicode-alpha-char-p`
`unicode-lower-case-p`
`unicode-upper-case-p`

**unicode-char-equal**

Function

Summary

Compares two characters, ignoring case using specified Unicode rules.

Package

lispworks

Signature

`unicode-char-equal char1 char2 &key style => flag`

Arguments

`char1`  A character
`char2`  A character
`style`  A keyword

Values

`flag`  A generalized boolean
The function `unicode-char-equal` returns true if the characters `char1` and `char2` are equal, ignoring case using Unicode rules specified by `style`.

The current implementation only supports one style of comparison:

`:simple-case-fold`

Compares characters using Unicode's simple case folding rules.

See also
- `unicode-char-greaterp`
- `unicode-char-lessp`
- `unicode-char-not-equal`

---

### Function

**unicode-char-greaterp**

**Summary**

Compares two characters, ignoring case using specified Unicode rules.

**Package**

`lispworks`

**Signature**

`unicode-char-greaterp char1 char2 &key style => flag`

**Arguments**

- `char1` A character
- `char2` A character
- `style` A keyword

**Values**

- `flag` A generalized boolean

**Description**

The function `unicode-char-greaterp` returns true if the character `char1` is greater than the character `char2`, similarly to `cl:char-greaterp` but ignoring case using Unicode rules specified by `style`.

The current implementation only supports one style of comparison:
The LISPWORKS Package

:simple-case-fold

Compares characters using Unicode’s simple lowercase folding rules.

See also

unicode-char-equal
unicode-char-not-greaterp

unicode-char-lessp

Function

Summary

Compares two characters, ignoring case using specified Unicode rules.

Package

lispworks

Signature

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 function unicode-char-lessp returns true if the character char1 is less than the character char2, similarly to 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 Unicode’s simple lowercase folding rules.

See also

unicode-char-equal
unicode-char-not-lessp
**unicode-char-not-equal**

*Function*

**Summary**

Compares two characters, ignoring case using specified Unicode rules.

**Package**
lispworks

**Signature**

`unicode-char-not-equal char1 char2 &key style => flag`

**Arguments**

- `char1`: A character
- `char2`: A character
- `style`: A keyword

**Values**

- `flag`: A generalized boolean

**Description**

The function `unicode-char-not-equal` returns true if the characters `char1` and `char2` are not equal, ignoring case using Unicode rules specified by `style`.

The current implementation only supports one style of comparison:

- `:simple-case-fold`
  
  Compares characters using Unicode's simple case folding rules.

**See also**

`unicode-char-equal`

---

**unicode-char-not-greaterp**

*Function*

**Summary**

Compares two characters, ignoring case using specified Unicode rules.

**Package**
lispworks

**Signature**

`unicode-char-not-greaterp char1 char2 &key style => flag`
The function `unicode-char-not-greaterp` returns true if the character `char1` is not greater than the character `char2`, similarly to `cl:char-not-greaterp` but ignoring case using Unicode rules specified by `style`.

The current implementation only supports one style of comparison:

`:simple-case-fold`

Comparing characters using Unicode’s simple lowercase folding rules.

See also

- `unicode-char-equal`
- `unicode-char-greaterp`

---

**unicode-char-not-lessp**

**Function**

Summary

Compares two characters, ignoring case using specified Unicode rules.

Package

`lispworks`

Signature

`unicode-char-not-lessp char1 char2 &key style => flag`

Arguments

- `char1` A character
- `char2` A character
- `style` A keyword

Values

- `flag` A generalized boolean
The function `unicode-char-not-lessp` returns true if the character `char1` is not less than the character `char2`, similarly to `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 Unicode's simple lowercase folding rules.

**See also**

* `unicode-char-equal`
* `unicode-char-lessp`

### `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 Unicode's "general category" for `char`. 
See also

`unicode-both-case-p`
`unicode-upper-case-p`

Function

### unicode-string-equal

**Summary**

Compares two strings, ignoring case using specified Unicode rules.

**Package**

`lispworks`

**Signature**

```
unicode-string-equal string1 string2 &key start1 start2 end1 end2 style => flag
```

**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**

- `flag` A generalized boolean

**Description**

The function `unicode-string-equal` compares 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` is true if the strings are equal and false otherwise.

The current implementation only supports one style of comparison:
- `:simple-case-fold`
  
  Compares each character of the strings using Unicode's simple case folding rules.
See also
choose-unicode-string-hash-function
unicode-string-not-equal

unicode-string-greaterp

Function

Summary
Compares two strings, ignoring case using specified Unicode rules.

Package
lispworks

Signature
unicode-string-greaterp 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 function unicode-string-greaterp compares the designated substrings of string1 and string2, similarly to cl:string-greaterp but ignoring case using Unicode rules specified by style. The values of start1 and start2 default to 0, while the values of end1 and end2 default to nil.

The value of mismatch-index is the index where the strings mismatch (as an offset from the beginning of string1) if substring1 is greater than substring2, or nil otherwise.

The current implementation only supports one style of comparison:
:simple-case-fold
Compares each character of the string using Unicode's simple lowercase folding rules.

See also
- `unicode-string-equal`
- `unicode-string-lessp`

### `unicode-string-lessp` Function

**Summary**
Compares two strings, ignoring case using specified Unicode rules.

**Package**
lispworks

**Signature**
`unicode-string-lessp string1 string2 &key start1 start2 end1 end2 style => mismatch-index`

**Arguments**
- `string1` A string designator
- `string2` A string designator
- `start1, end1` Bounding index designators of `string1`
- `start2, end2` Bounding index designators of `string2`
- `style` A keyword

**Values**
- `mismatch-index` A bounding index of `string1` or `nil`

**Description**
The function `unicode-string-lessp` compares the designated substrings of `string1` and `string2`, similarly to `cl:string-greaterp` but ignoring case using Unicode rules specified by `style`. The values of `start1` and `start2` default to 0, while the values of `end1` and `end2` default to `nil`.

The value of `mismatch-index` is the index where the strings mismatch (as an offset from the beginning of `string1`) if `substring1` is 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 Unicode's simple lowercase folding rules.

See also  
unicode-string-equal
unicode-string-greaterp

**unicode-string-not-equal**

*Function*

**Summary**
Compares two strings, ignoring case using specified Unicode rules.

**Package**
lispworks

**Signature**

`unicode-string-not-equal 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 function `unicode-string-not-equal` compares 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 value of `mismatch-index` 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 string using Unicode’s simple case folding rules.

See also  `unicode-string-equal`

---

**unicode-string-not-greaterp**

*Function*

**Summary**

Compares two strings, ignoring case using specified Unicode rules.

**Package**

`lispworks`

**Signature**

`unicode-string-not-greaterp 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 function `unicode-string-not-greaterp` compares the designated substrings of `string1` and `string2`, similarly to `cl:string-not-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 not greater than `substring2`, or `nil` otherwise.
The current implementation only supports one style of comparison:

:simple-case-fold

    Compares each character of the string using Unicode's simple lowercase folding rules.

See also

unicode-string-equal
unicode-string-greaterp

**unicode-string-not-lessp**

**Function**

**Summary**

Compares two strings, ignoring case using specified Unicode rules.

**Package**

lispworks

**Signature**

```lisp
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 function `unicode-string-not-lessp` compares the designated substrings of `string1` and `string2`, similarly to `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 less than substring2, or nil otherwise.

The current implementation only supports one style of comparison:

:simple-case-fold

Compared each character of the string using Unicode's simple lowercase folding rules.

See also

unicode-string-equal
unicode-string-lessp

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 Unicode's "general category" for char.
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.
Note: 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.

Note: while product can in principle be any Lisp object, values of product are compared by eq, so you should use keywords.

Note: The CAPI provides a way to store window geometry - see the entry for capi:top-level-interface-save-geometry-p in the LispWorks CAPI Reference Manual.

Example

This example is on Microsoft Windows. Note the use of backslashes as directory separators in the path argument:

```
(setf (user-preference "My Stuff\FAQ"
       "Ultimate Answer"
       :product :deep-thought)
    42)
=> 42
```

This is equivalent to the previous example, and is portable because we avoid the explicit directory separators in the path argument:

```
(setf (user-preference (list "My Stuff" "FAQ")
       "Ultimate Answer"
       :product :deep-thought)
    42)
=> 42
```

We can retrieve values on Windows like this:

```
/user-preference "My Stuff\FAQ"
"Ultimate Answer"
:product :deep-thought)
```

```
=> 42
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.

Description
This macro executes the body of code if the form evaluates to a non-nil result. Within the body, the variable var is bound to the result of the form.

Example
The form

(when-let (position (search string1 string2))
  (print position))

macroexpands to
(let ((position (search string1 string2)))
  (when position
    (print position)))

See also when-let*

when-let*

Macro

Summary
Executes a body of code if a series of forms evaluates to non-nil, propagating the results of the forms through the body of code.

Package lispworks

Signature when-let* bindings &body body => result
bindings ::= ((var form)*)

Arguments
var A variable whose value is used in the evaluation of body.
form A form, which must evaluate to non-nil.
body A body of code to be evaluated conditionally on the result of form.

Values
result The result of evaluating body using the value var.

Description
The macro when-let* expands into nested when-let forms. The bindings are evaluated in turn as long as each returns non-nil. If the last binding evaluates to non-nil, body is executed. Within the code body, each variable var is bound to the result of the corresponding form form.
Example
(defmacro divisible (n &rest divisors)
  `(when-let* ,(loop for div in divisors
          collect (list (gensym)
                       (zerop (mod n div)))))
  t))

See also when-let

whitespace-char-p

Summary Tests whether a character represents white space.

Package lispworks

Signature whitespace-char-p char => bool

Arguments char A character.

Values bool t if char represents white space; nil otherwise.

Description This predicate recognizes [whitespace1], as described in the standard:

“Space and non-graphic characters that only moved the print position.”

If sys:*extended-spaces* is t, 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, signalling errors and then continuing to the next action item.
The with-action-item-error-handling macro executes the body with action-list-var and action-item-var are bound to the action list and item respectively. If ignore-errors-p is set to t then errors are handled. The behavior of the handler is to signal a warning in which the action-list, item and original error are all reported; execution then continues with the next action-item.

Example

```lisp
(defun my-execution-function (the-action-list
  other-args
  &key ignore-errors-p
  &allow-other-keys)
  (with-action-list-mapping (the-action-list
    an-action-item
    action-item-data)
    (with-action-item-error-handling (the-action-list
      an-action-item
      ignore-errors-p)
      (do-something-interesting-first)
      (apply (car action-item-data) other-args (cdr
        action-item-data)))))
```

If this function was invoked with the keyword argument :ignore-errors-p t, and an error was signalled while executing 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 signalled and execution would continue with the next action-item.

See also *handle-missing-action-in-action-list*

**with-action-list-mapping**

*Macro*

**Summary**
Maps over an action list’s actions with given variables bound to the executing action and its data.

**Package**
lispworks

**Signature**

```
with-action-list-mapping action-list item-var data-var
&optional post-process &body body)
```

**Arguments**

- `action-list`: An action list.
- `item-var`: A Lisp symbol.
- `data-var`: A Lisp symbol.
- `post-process`: A keyword.
- `body`: A body of Lisp code.

**Description**
The with-action-list-mapping macro 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
34 The LISPWORKS Package
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 15, “Multiprocessing”.

**allowing-block-interrupts**  
*Macro*

**Summary**  
Allows control over blocking interrupts.

**Signature**  
`allowing-block-interrupts start-blocked &body body => results`

**Package**  
`mp`

**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-process-`
unblock-interrupts and current-process-unblock-interrupts.

Within the dynamic scope of allowing-block-interrupts, you can switch the blocking of interrupts on and off. Blocking interrupts prevents any interruption of the current process, including process-interrupt, process-kill, process-reset, process-break and process-stop. These interrupts are all queued and processed once interrupts become unblocked.

Blocking interrupts also blocks interrupts due to 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

### 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**

make-barrier

### barrier-change-count

**Function**

**Summary**

Changes the count of a barrier.

**Package**

mp
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

**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.
The count value can be changed by `barrier-unblock`, `barrier-enable`, `barrier-disable` or `barrier-change-count`.

See also `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-enable**  
*Function*

**Summary**  
Ensures that a barrier is enabled.

**Package**  
`mp`

**Signature**  
`barrier-enable barrier count &optional kill-waiting`
### barrier-enable

**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-unblock`

### barrier-name

**Function**

- **Summary**: Returns the name of the barrier
- **Package**: `mp`
- **Signature**: `barrier-name barrier => name`
- **Arguments**: *barrier*: A barrier.
- **Values**: *name*: A string.
- **Description**: The function `barrier-name` returns the name of the barrier, as supplied or defaulted in the call to `make-barrier`.
- **See also**: `make-barrier`

### barrier-pass-through

**Function**

- **Summary**: Increments the arriver count of a barrier.
Package: mp

Signature: barrier-pass-through barrier => result

Arguments: barrier A barrier.

Values: result One of the keyword symbols :unblocked and :passed-through.

Description: The function barrier-pass-through increments the arriver count of the barrier barrier. If the arriver count thereby reaches the count, barrier-pass-through unblocks the barrier and returns :unblocked, otherwise it returns :passed-through.

barrier-pass-through is equivalent to calling barrier-wait with pass-through t. See barrier-wait for details.

See also: barrier-wait

barrier-unblock

Function

Summary: Unblocks a barrier.

Package: mp

Signature: barrier-unblock barrier &key disable reset-count killwaiting

Arguments: disable A boolean.
reset-count A positive fixnum, t or nil.
killwaiting 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-kill`).

See also
- `process-kill`
- `barrier-wait`

### `barrier-wait` Function

**Summary**
Waits on a barrier until enough threads arrive.

**Package**
mp

**Signature**
`barrier-wait barrier &key timeout callback pass-through discount-on-abort discount-on-timeout disable-on-unblock => result`

**Arguments**
- `barrier` A barrier.
- `timeout` A non-negative 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  One of the keyword symbols :unblocked, :passed-through, :timeout, nil or t.

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 if 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-kill or throwing using process-interrupt), 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 \texttt{barrier-wait} will wait, as if the barrier had just been created.

4. If \texttt{disable-on-unblock} is true, \texttt{barrier-wait} then disables the barrier. That means that until it is enabled, any call to \texttt{barrier-wait} will return immediately.

5. It wakes up all the waiting threads.

6. It returns the symbol \texttt{:unblocked}.

The possible values of \texttt{result} occur in these circumstances:

\begin{itemize}
\item \texttt{t} \quad The current process waited and some other process unblocked the barrier.
\item \texttt{:unblocked} \quad The current process unblocked the barrier.
\item \texttt{:timeout} \quad The wait timed out.
\item \texttt{:passed-through} \quad Pass through because \texttt{pass-through} was true.
\item \texttt{nil} \quad The barrier is disabled.
\end{itemize}

\textbf{See also} \ \texttt{make-barrier}

\textbf{change-process-priority}

\textit{Function}

\textbf{Summary} \quad Changes the priority of a process.

\textbf{Package} \quad \texttt{mp}

\textbf{Signature} \quad \texttt{change-process-priority process new-priority => new-priority}

\textbf{Arguments} \quad \texttt{process} \quad 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 => signalledp

Arguments condvar A condition variable

Values signalledp 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.

The return value signalledp is non-nil if some processes were signalled, or nil if there were no processes waiting.

See also condition-variable-wait make-condition-variable

condition-variable-signal Function

Summary Wakes one thread waiting on a given condition variable.
Package  mp

Signature   condition-variable-signal  condvar => signalledp

Arguments  condvar         A condition variable

Values     signalledp     A generalized boolean

Description  The function condition-variable-signal wakes exactly one thread waiting on the condition variable condvar. In most uses of condition variables, the caller should be holding the lock that the waiter used when calling condition-variable-wait for this condition variable, but this is not required.

The return value signalledp is non-nil if a process was signalled, or nil if there were no processes waiting.

See also  condition-variable-wait
          make-condition-variable

condition-variable-wait  Function

Summary  Waits for a given condition variable to be signalled.

Package  mp

Signature  condition-variable-wait  condvar  lock  &key  timeout  wait-reason  =>  wakep

Arguments  condvar         A condition variable

lock         A mp:lock

timeout      nil or a positive real

wait-reason   A string

Values     wakep         A generalized boolean
Description

The function `condition-variable-wait` waits at most `timeout` seconds for the condition variable `condvar` to be signalled. 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 `wakeup` 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.

Notes

`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`

**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

create-simple-process  
Function

Summary Creates and returns a simple process, which is a process with no stack of its own.

Signature create-simple-process name function wait-function &key function-arguments wait-function-arguments priority => process

Package mp

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A string or symbol</td>
</tr>
<tr>
<td>function</td>
<td>A function</td>
</tr>
<tr>
<td>wait-function</td>
<td>A function</td>
</tr>
<tr>
<td>function-arguments</td>
<td>A list of arguments for function</td>
</tr>
<tr>
<td>wait-function-arguments</td>
<td>A list of arguments for wait-function</td>
</tr>
<tr>
<td>priority</td>
<td>A fixnum</td>
</tr>
</tbody>
</table>

Values process A simple process

Description The create-simple-process function creates and returns a simple process, which is a process that has no stack of its own.

The name argument is a string or symbol that names the process. The function argument is a function to be run in the process, and the wait-function argument is a wait function that determines when the process function is run. The value of function-arguments is a list of arguments to which the process function is applied. The value of wait-function-arguments is a list of arguments to which the wait function is applied. The
The `:priority` argument is a fixnum that specifies a priority for the process. The default priority is the value of `*default-simple-process-priority*`, and is usually 0.

When the `wait` function, applied to the `wait-function-arguments`, returns a value other than `nil`, the `process` function is applied to the function-arguments. The `process` function is executed inside an `mp:without-preemption` form. If an error occurs in a simple process, that process is stopped and a continuable error is signaled in the process that was running at the time the simple process was started (or the last process to run if the system was idle). Continuing from the error restarts the simple process.

Because a simple process has no stack of its own, it can be executed on an arbitrary stack. However, simple processes have restrictions, the primary one being that they cannot block. The following interfaces cannot be used in a simple process:

- `mp:mailbox-read` (with an empty mailbox)
- `mp:process-allow-scheduling`
- `mp:process-lock`
- `mp:process-wait`
- `mp:process-wait-with-timeout`
- `cl:sleep`
- `mp:sleep-for-time`
- `mp:wait-for-mailbox`
- `mp:wait-processing-events`
- `mp:with-lock`
- CAPI functions that block
Other Common Lisp functions might not work if they attempt to block. This applies in particular to I/O functions on streams such as pipes and to \texttt{(setf gethash)} on a hash table that another process is mapping over.

For more information, see Chapter 15, “Multiprocessing”.

**Example**

The following example creates a simple process that prints the value of \texttt{*a*} to the background output when the value is other than \texttt{nil}. The process function then sets \texttt{*a*} to \texttt{nil}. From a listener, the value of \texttt{*a*} can be set to trigger the process to run once and then sleep again.

\begin{verbatim}
(defun a ()
  (let ((a *a*))
    (setq *a* nil)
    (format mp:*background-standard-output*
      "*a* is -a-\$a\$ " a)))
\end{verbatim}

\begin{verbatim}
(defun b () *a*)
\end{verbatim}

\begin{verbatim}
(setq r (mp:create-simple-process 'test-proc 'a 'b))
\end{verbatim}

\begin{verbatim}
#<MP::SIMPLE-PROCESS Name TEST-PROC Priority 0 State NIL>
\end{verbatim}

See also \texttt{process-run-function}

\texttt{*current-process*} \hspace{1cm} \textit{Variable}

**Summary**

Contains the object that is the current process.

**Package** \hspace{1cm} \texttt{mp}
**The MP Package**

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.

**Signature**

\[
\text{current-process-block-interrupts} \Rightarrow t
\]

**Description**

The function `current-process-block-interrupts` blocks interrupts in the current process.

It signals an error if called outside the dynamic scope of `allowing-block-interrupts` or `with-interrupts-blocked`.

Blocking interrupts prevents any interruption of the current process, including `process-interrupt`, `process-kill`, `process-reset`, `process-break` and `process-stop`. These interrupts are all queued and processed once interrupts become unblocked.

Blocking interrupts also blocks interrupts due to 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-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 in 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 process being poked in some way.

**Example**

Supposed you want to have a process that each minute does some cleanup, but may also be told by other processes to go and do the cleanup. The process be doing:

```
(loop
  (mp:current-process-pause 60 'check-for-need-cleanup)
  (do-cleanup))
```

Another process which wants to provoke a cleanup will do:

```
(setup-cleanup-flag)

(mp:process-poke *cleanup-process*)
```

Note that `check-for-need-cleanup` is passed to `current-process-pause`, because another process may call `process-poke` after `current-process-pause` was called but before it went to sleep. If `check-for-need-cleanup` was not passed, `current-process-pause` would unnecessarily sleep the whole 60 seconds in this case. The same thing could be implemented by `process-wait-with-timeout`, but the implementation above does not require a wait function that can run in another dynamic scope repeatedly at arbitrary times, and it uses much less system resources. It is also easier to debug.

**See also**

`process-poke`

---

### current-process-unblock-interrupts

**Function**

**Summary**

Unblocks interrupts in the current process.

**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

Summary

Debug a thread other than the current process.

Package

mp

Signature

debug-other-process process

Arguments

process A process or a string.

Description

The function mp:debug-other-process causes the debugger to be entered in the given process. If process is a string, the process is found as if by mp: find-process-from-name. The list of process names can be found via mp:ps.

See also

find-process-from-name
ps
**default-process-priority**  
*Variable*

Summary  
The default priority for processes.

Package  
mp

Description  
The *default-process-priority* variable contains the default priority for processes.

See also  
process-run-function  
create-simple-process  
*default-simple-process-priority*

---

**default-simple-process-priority**  
*Variable*

Summary  
The default priority for simple processes.

Package  
mp

Description  
The *default-simple-process-priority* variable contains the default priority for simple processes.

See also  
create-simple-process  
*default-process-priority*

---

**ensure-process-cleanup**  
*Function*

Summary  
Run forms when a given process terminates.

Package  
mp

Signature  
ensure-process-cleanup cleanup-form &optional process =>

Arguments  
cleanup-form  
Form to run when process terminates.
The process to watch for termination. By default, this is the value of *current-process*.

Values

None.

Description

Ensures that the cleanup-form is present for the given process. When the process terminates, its cleanup forms are run. Cleanup forms can be functions of one argument (the process), or lists, in which case the car is applied to the process and the cdr of the list.

When adding cleanup forms, this function uses equal to ensure that the form is only added once.

Notes

You can test for whether the current process is executing its cleanups with current-process-in-cleanup-p.

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-kill

find-process-from-name

Function

Summary

Finds a process from its name.

Package

mp
Signature  
\textbf{find-process-from-name} \textit{process-name} \texttt{=>} \textit{result}

Arguments  
\textit{process-name} \quad \text{A string.}

Values  
\textit{result} \quad \text{A \texttt{mp:process}, or \texttt{nil}.}

Description  
The function \textbf{find-process-from-name} returns the process with the name \textit{process-name}. If there is no such process, the function returns \texttt{nil}.

Example  
\texttt{CL-USER 16 > (mp:find-process-from-name "Listener 1")}
\texttt{#<MP:PROCESS Name "Listener 1" Priority 600000 State "Running">}

See also  
\texttt{get-process}

general-handle-event  \quad \textit{Generic function}

Summary  
"handles" an event, depending on the type of the event object.

Package  \quad \texttt{mp}

Signature  
\textbf{general-handle-event} \textit{event-object}

Arguments  
\textit{event-object} \quad \text{A Lisp object.}

Description  
The generic function \textbf{general-handle-event} "handles" the \textit{event-object}. What this actually means depends on the type of the object.

There are system defined methods for these classes:

\begin{itemize}
  \item \texttt{list} \quad Apply the \texttt{car} to the \texttt{cdr}.
  \item \texttt{function} \quad Call it.
  \item \texttt{symbol} \quad If fbound call it, otherwise do nothing.
  \item \texttt{t} \quad Do nothing.
\end{itemize}
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`

---

**get-current-process**

*Function*

**Summary**

Returns the current Lisp process.

**Package**

`mp`

**Signature**

`get-current-process => result`

**Values**

`result`  
A `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 `*current-process*`, which can be bound to another process. In particular, when a process A calls the `wait-function` of process B, in the `wait-function` `get-current-process` returns the process A, but `*current-process*` is bound to process B.  

`result` is `nil` if multiprocessing is off.

See also `*current-process*`

---

**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`
The MP Package

**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`.
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 an `mp:process` object for the initial thread using the arguments in that list as if in the call

```lisp
(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.

**Note:** 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`

```lisp
(defvar *initial-processes* nil)
```

**Package**

`mp`
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.

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

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

<table>
<thead>
<tr>
<th><strong>Function</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists all the Lisp processes currently in the system.</td>
</tr>
</tbody>
</table>

| **Summary** |
| Lists all the Lisp processes currently in the system. |

| **Package** |
| mp |

| **Signature** |
| list-all-processes => process-list |

| **Arguments** |
| None. |

| **Values** |
| process-list |
| A list of all the currently active Lisp processes. |

| **Description** |
| Returns a list of all the active Lisp processes in LispWorks. |
Example

```
CL-USER 71 > (pprint (mp:list-all-processes))

(#<MP:PROCESS Name "Editor 1" Priority 70000000 State "Waiting for events">
 #<MP:PROCESS Name "Listener 1" Priority 70000000 State "Running">
 #<MP:PROCESS Name "LispWorks 5.1.0" Priority 70000000 State "Waiting for events">
 #<MP:PROCESS Name "default listener process" Priority 60000000 State "Waiting for terminal input.">
 #<MP:PROCESS Name "CAPI Execution Listener 1" Priority 60000000 State "Running">
 #<MP:PROCESS Name "Background execute 2" Priority 50000000 State "Waiting for job to execute">
 #<MP:PROCESS Name "Background execute 1" Priority 50000000 State "Waiting for job to execute">
 #<MP:PROCESS Name "Editor DDE server" Priority 0 State "Waiting for an event">
 #<MP:PROCESS Name "The idle process" Priority -536870912 State "Running (preempted)">)
```

lock-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.

**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 if 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 if 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`.

Notes `lock-recursive-p` does not check whether the lock is currently locked recursively. The function `lock-recursively-locked-p` does that.

See also `make-lock`
Package: mp

Signature: lock-name lock => name

Arguments:
lock: A lock object

Values:
name: A string

Description: The lock-name function 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 lock-owner function 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 whilst holding the lock) then `result` is `:unknown`.

`result` is `nil` if lock is not locked.

Example

```lisp
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-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 generalized boolean

Description

The `mailbox-empty-p` function returns `t` if the given `mailbox` is empty and `nil` otherwise.

See also

`mailbox-send`  
`mailbox-peek`  
`mailbox-read`  
`make-mailbox`

---

**Function**

**mailbox-peek**

Summary

Peeks at the first object in a mailbox.

Package

`mp`

Signature

`mailbox-peek mailbox => result`

Arguments

*mailbox*    A mailbox.

Values

*result*    Any object or `nil`.

Description

The `mailbox-peek` function returns the first object in the mailbox without removing it. If the mailbox is empty, `nil` is returned.

See also

`mailbox-empty-p`  
`mailbox-send`  
`mailbox-read`  
`make-mailbox`
**mailbox-read**  

**Function**

**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 `mailbox-read` function 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 (or the string "Waiting for message" if `wait-reason` is `nil`) and the `timeout` argument are both passed to `process-wait-with-timeout`.

The default value of `wait-reason` is `nil` and the default value of `timeout` is `nil`.

**See also**
- `mailbox-empty-p`
- `mailbox-peek`
- `mailbox-send`
- `mailbox-wait-for-event`
make-mailbox
process-wait-with-timeout

**mailbox-reader-process**  
*Function*

**Summary**
Returns the reader process of the mailbox.

**Package**
*mp*

**Signature**
`mailbox-reader-process mailbox => process`

**Arguments**
`mailbox`
A mailbox

**Values**
`process`
A process

**Description**
The `mailbox-reader-process` function 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 `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-for-event

Function

Summary
Waits for an event in a "windowing friendly" way.

Package
mp

Signature
mailbox-wait-for-event mailbox &key wait-reason wait-function process-other-messages-p no-hang-p stop-at-user-operation-p => event

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 event 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.

wait-function is the wait function to be used, which is called with the mailbox mailbox as its argument. If wait-function is not supplied, a function that returns t when the mailbox is not empty is used internally.

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-wait`

### make-condition-variable

**Function**

**Summary**
Makes a condition variable.

**Package**
`mp`

**Signature**
`make-condition-variable &key name => condvar`

**Arguments**
- `name` A string naming the condition variable.

**Values**
- `condvar` A condition variable.
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.

**Values**

- `lock` The lock object.

**Description**

`make-lock` creates a lock object. See “Locks” on page 179 for a general description of locks.

`name` names the lock and can be queried with `mp:lock-name`. The default value of `name` is "Anon".
important-p controls whether the lock is automatically freed when the holder process finishes. When important-p is true, the system notes that this lock is important, and automatically frees it when the holder process finishes. important-p should be nil for locks which are managed completely by the application, as it is wasteful to record all locks in a global list if there is no need to free them automatically. This might be appropriate when two processes sharing a lock must both be running for the system to be consistent. If one process dies, then the other one kills itself. Thus the system does not need to worry about freeing the lock because no process could be waiting on it forever after the first process dies. The default value of important-p is nil.

safep controls whether the lock is safe. A safe lock gives an error if process-unlock is called on it when it is not locked by the current process, and potentially in other 'dangerous' circumstances. An unsafe lock does not signal these errors. The default value of safep is t.

recursivep, when true, allows 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 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 recursivep is t.

sharing, when true, causes lock to be a “sharing” lock object, which supports sharing and exclusive locking. At any given time, an 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

- create-simple-process
- lock-recursive-p
- process-lock
- process-unlock
- schedule-timer
- with-lock

**make-mailbox**

Function

**Summary**

Make a new mailbox for the current process.

**Package**

mp

**Signature**

```lisp
make-mailbox &key size => mailbox
```

**Arguments**

- `size` An integer

**Values**

- `mailbox` A mailbox

**Description**

The function `make-mailbox` returns a new mailbox.
size specifies the initial size of the mailbox.

The reader process is set to the current process.

See also

mailbox-empty-p
mailbox-peek
mailbox-read
mailbox-send

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 make-named-timer function creates and returns a named timer. The first argument is a string or symbol naming the timer. The second argument is a function to be applied to the remaining arguments when the timer expires. Use the function schedule-timer or schedule-timer-relative to set an expiration time.

In comparison, the function make-timer creates an unnamed timer.

Example

(setq timer (mp:make-named-timer 'timer-1 'print 10 *standard-output*))

#<Time Event TIMER-1 : PRINT>
See also  make-timer
          schedule-timer
          schedule-timer-milliseconds
          schedule-timer-relative
          schedule-timer-relative-milliseconds
          timer-expired-p
          timer-name
          unschedule-timer

make-semaphore  Function

Summary  Makes a semaphore.

Package  mp

Signature  make-semaphore &key name count => sem

Arguments  name  An object.
            count  A non-negative fixnum.

Values  sem  A semaphore.

Description  The function make-semaphore returns a new semaphore for use with semaphore-acquire and semaphore-release. The unit count is initialized to count, which defaults to 1. If name is supplied, the semaphore will have that name. If name is not supplied, the semaphore will be given a unique anonymous name.

See also  semaphore-acquire
          semaphore-count
          semaphore-name
          semaphore-release
          semaphore-wait-count
### make-timer 
**Function**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Creates and returns an unnamed timer.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td><code>make-timer function &amp;rest arguments =&gt; timer</code></td>
</tr>
<tr>
<td>Package</td>
<td><code>mp</code></td>
</tr>
</tbody>
</table>
| Arguments     | `function` A function  
`arguments` A set of arguments to `function` |
| Values        | `timer` A timer |
| Description   | The `make-timer` function 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       | `(setq timer  
  (mp:make-timer 'print 10 *standard-output*))  
=>  
#<Time Event : PRINT>` |
| See also      | `make-named-timer`  
`make-timer`  
`schedule-timer`  
`schedule-timer-milliseconds`  
`schedule-timer-relative`  
`schedule-timer-relative-milliseconds`  
`timer-expired-p`  
`timer-name`  
`unschedule-timer` |
map-all-processes

Function

Summary Calls a predicate function on processes in turn until a true value is returned.

Package mp

Signature map-all-processes function =>

Arguments function A function taking one argument

Values None.

Description The function map-all-processes calls function on processes in the image, including simple processes.

function is passed each process in turn as its single argument.

If function returns false, map-all-processes calls function on the next process.

If function returns true, map-all-processes returns immediately, 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 `map-all-processes-backtrace` function 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*

<table>
<thead>
<tr>
<th><strong>Summary</strong></th>
<th>Produces a backtrace for a process</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package</strong></td>
<td><code>mp</code></td>
</tr>
<tr>
<td><strong>Signature</strong></td>
<td><code>map-process-backtrace process function</code></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td></td>
</tr>
<tr>
<td><code>process</code></td>
<td>A process</td>
</tr>
<tr>
<td><code>function</code></td>
<td>A function taking one argument</td>
</tr>
<tr>
<td><strong>Values</strong></td>
<td>None.</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The <code>map-process-backtrace</code> function collects a backtrace for the process specified by <code>process</code>, and the function <code>function</code> is called on each line of the backtrace in turn.</td>
</tr>
</tbody>
</table>
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 the function for all non-simple processes.

**Package**

`mp`

**Signature**

`map-processes function`

**Arguments**

`function` A function taking one argument

**Values**

None.

**Description**

The function `map-processes` calls `function` for every non-simple process.

`function` is passed each such process as its single argument.

See also

`map-all-processes`
**notice-fd**  
*Function*

**Summary**
Add a file descriptor to the set of interesting input file descriptors.

**Package**
mp

**Signature**
notice-fd fd

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd</td>
<td>A UNIX file descriptor</td>
</tr>
</tbody>
</table>

**Values**
None.

**Description**
The `notice-fd` function adds the given `fd` to the set of fds that cause LispWorks to wake up when they contain input.

This function is not implemented on Microsoft Windows.

**See also**
unnotice-fd

---

**process-alive-p**  
*Function*

**Summary**
Determines if a process is alive.

**Package**
mp

**Signature**
process-alive-p process => bool

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>process</td>
<td>A process</td>
</tr>
</tbody>
</table>

**Values**
bool  
A boolean

**Description**
The `process-alive-p` function returns t if `process` is alive, that is, if `mp:process-kill` has not been called on the process.

**Example**

```
(mp:process-alive-p mp:*current-process*)
```
(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
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 note
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  

<table>
<thead>
<tr>
<th>Summary</th>
<th>Breaks a Lisp process and enters the debugger.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>mp</td>
</tr>
<tr>
<td>Signature</td>
<td>process-break process =&gt;</td>
</tr>
<tr>
<td>Arguments</td>
<td>process</td>
</tr>
<tr>
<td>Values</td>
<td>None.</td>
</tr>
<tr>
<td>Description</td>
<td>The function process-break forces the process process to break and enter the debugger.</td>
</tr>
</tbody>
</table>

process-continue  

<table>
<thead>
<tr>
<th>Summary</th>
<th>Wakes up a process.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>mp</td>
</tr>
<tr>
<td>Signature</td>
<td>process-continue process =&gt; nil</td>
</tr>
<tr>
<td>Arguments</td>
<td>process</td>
</tr>
<tr>
<td>Description</td>
<td>The function process-continue is wakes up the process process, regardless of whether it is sleeping, stopped or waiting. process-continue returns nil.</td>
</tr>
</tbody>
</table>

process-exclusive-lock  

<table>
<thead>
<tr>
<th>Summary</th>
<th>Like process-lock, but on a &quot;sharing&quot; lock.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>mp</td>
</tr>
</tbody>
</table>
Signature  
\texttt{process-exclusive-lock} \hspace{1em} \texttt{sharing-lock} \hspace{1em} \&\texttt{optional} \hspace{1em} \texttt{whostate} \\
\hspace{1em} \texttt{timeout} 

Arguments  
\textit{sharing-lock} \hspace{1em} A sharing lock. \\
\textit{whostate} \hspace{1em} The status of the process while the lock is 
\hspace{1em} locked, as seen in the Process Browser. \\
\textit{timeout} \hspace{1em} A timeout interval, in seconds. 

Description  
The function \texttt{process-exclusive-lock} is the same as \texttt{process-lock}, but on a "sharing" lock. It waits until the lock is 
\hspace{1em} free before locking in exclusive mode. 

Calls to \texttt{process-exclusive-lock} should be paired with 
\texttt{process-exclusive-unlock} calls. In most cases the macro 
\texttt{with-exclusive-lock} the best way to achieve this. 

Notes  
It is not possible to use exclusive lock in the scope of a shar-
\hspace{1em} ing lock on the same lock, and trying to do this will cause the 
\hspace{1em} process to hang. Whether it is possible to use an exclusive 
\hspace{1em} lock inside an exclusive lock of the same lock is determined 
\hspace{1em} by the \texttt{recursivep} argument in \texttt{make-lock}. 

\texttt{process-exclusive-lock} is guaranteed to return if it locked 
\hspace{1em} process, but may throw before locking, as described in 
\hspace{1em} “Guarantees and limitations when locking and unlocking” 
\hspace{1em} on page 181. 

See also  
\texttt{make-lock} \\
\texttt{process-lock} \\
\texttt{with-exclusive-lock} 

\texttt{process-exclusive-unlock}  

\textit{Function}  

Summary  
Like \texttt{process-unlock}, but on a "sharing" lock. 

Package  
\texttt{mp}
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.

`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 181.

See also `process-exclusive-lock` `process-unlock` `with-exclusive-lock`
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:

```lisp
(defvar *binding-1* 10)

(let ((mp:*process-initial-bindings* (cons '(*binding-1* . 20) mp:*process-initial-bindings*))
     (mp:process-run-function '*binding-1*
      ()
      #'(lambda (stream)
          (format stream "&Binding 1 is ~S.~%" *binding-1*))
      *standard-output*)
     (sleep 1))
=>
Binding 1 is 20.
```

This example shows the special case with an unbound symbol:

```lisp
(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.~%" *binding-2*)))))
          (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.~%" *binding-2*)))))
          (check-binding-2))
        (flet ((check-binding-2 (let (*binding-2* 123))
                  (check-binding-2))))
        =>' Binding 2 is unbound.
        Binding 2 is 123.

process-interrupt

Function

Summary
Interrupts a process.

Package
mp

Signature
process-interrupt process function &rest arguments =>

Arguments
process A process.
function A function to apply on resuming process.
arguments Arguments to supply to function.

Values
None.

Description
Causes the Lisp process process to apply function to arguments when it is next resumed. Afterwards the process resumes its normal execution. A waiting process is temporarily woken up.
**process-join**  
*Function*

**Summary**  
Waits until a specified process dies, or a *timeout* is reached.

**Package**  
`mp`

**Signature**  
```
process-join process &key timeout => flag
```

**Arguments**  
- `process`  
  A process.
- `timeout`  
  A non-negative number.

**Values**  
- `flag`  
  A boolean.

**Description**  
The function `process-join` waits until the process `process` dies, or `timeout` seconds passed.

If the process dies then `process-join` returns `t`. If the timeout passed it returns `nil`.

`process-join` can be used on dead processes, and in this case returns `t` immediately.

The effect of `process-join` is similar to

```
(mp:process-wait-with-timeout  
"Waiting for process to die" timeout  
#'(lambda (x)  
  (not (mp:process-alive-p x))) process)
```

but the call above may not return until the next time the scheduler runs, possibly causing a delay. In contrast `process-join` returns immediately when the process dies.

**See also**  
`process-wait-with-timeout`

---

**process-kill**  
*Function*

**Summary**  
Kills the specified Lisp process.
The MP Package

package mp

signature process-kill process =>

arguments process A process.

values None.

description The function process-kill kills the specified Lisp process.

see also ensure-process-cleanup

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 timeout. 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 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 181.

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
(setf 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 (setf (mp:process-mailbox mp:*current-process*)
(mp:make-mailbox))
**process-name**  
*Function*

Summary  
Returns the name of a specified process.

Package  
mp

Signature  
process-name process => name

Arguments  
process  
A process.

Values  
name  
The name of the process specified by process.

Description  
Returns the name of the specified Lisp process.

**process-p**  
*Function*

Summary  
A predicate to indentify non-simple processes

Package  
mp

Signature  
process-p object => bool

Arguments  
object  
Any object

Values  
bool  
A generalized boolean.

Description  
The process-p function returns t if object is a non-simple process, and nil otherwise.

**process-plist**  
*Function*

Summary  
Returns the plist associated with a process. This function is deprecated.

Package  
mp
process-plist

Signature
process-plist process => plist

Arguments
process A process

Values
plist A plist

Description
The process-plist function 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

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, process-poke causes it to run its wait-function as soon as possible, and if the wait function returns true, the process returns from process-wait.

This has an effect only in SMP LispWorks, where the running of the wait-function can happen asynchronously.

process-poke can be used to avoid delays that happen because the next execution of the wait-function does not happen immediately. Without the call to process-poke, the process may wake up after some delay.
process-poke returns t if it actually poked the process or nil otherwise (when the process is not waiting or is stopped).

Example

```lisp
(my-queue-an-event-for-the-workers)
(dolist (process *my-worker-processes*)
  (when (mp:process-poke process) (return))
)
```

See also

**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 mp:change-process-priority.

Example

```lisp
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
process-private-property

Signature

process-private-property indicator &optional default => result

(setq mp:process-private-property) value indicator &optional default => result

Arguments

indicator A Lisp object.

default A Lisp object.

Values

result value or default

Description

The function process-private-property gets or sets the value that is associated with indicator in the private properties of the current process (that is, the result of calling get-current-process).

If indicator is not associated with a value in the private properties, process-private-property returns default.

(setq process-private-property) overwrites any existing value for indicator.

The default value of default is nil.

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

(setq 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.

Example

```
(process-property 'foo (get-current-process) 'bar)
=> BAR
(setf (process-property 'foo) 'foo-value)
=> FOO-VALUE
(process-property 'foo)
=> FOO-VALUE
```

See also

- `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.
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.

True values specify that process should have a mailbox. A mailbox object is used as-is; a string is used as the name of a new mailbox; and t causes it to create a mailbox with the same name as process, that is, name.

Note that both process-send and process-wait-for-event force the relevant process to have a mailbox.

The new process is preset to apply function to arguments and runs in parallel, while process-run-function returns immediately.
Example

```
CL-USER 253 > (defvar *stream* *standard-output*)
*STREAM*

CL-USER 254 > (mp:process-run-function
  "My process"
  '(:priority 42)
  #'(lambda (x)
    (loop for i below x
      do (and (print i *stream*)
             (sleep 1))
    finally
      (print (mp:process-priority
               mp:*current-process*)
             *stream*)))))

 3)
#<MP:PROCESS Name "My process" Priority 850000 State
  "Running">

0
1
2
42
CL-USER 255 >
```

See also  
*default-process-priority*

---

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.
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`.

**Note:** 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 `process-send` function 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`.

`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.
35 The MP Package

- :default — set the priority to the normal running priority.

See also
- mailbox-send
- process-wait-for-event

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 process-sharing-lock should be matched by calls to process-sharing-unlock. Normally with-sharing-lock is the best way to achieve this.

Notes
It is possible to lock for sharing inside the scope of sharing lock and inside the scope of exclusive lock.

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 181.

See also process-lock
process-sharing-unlock
with-sharing-lock

**process-sharing-unlock**  
*Function*

Summary  
Removes a sharing lock.

Package  
mp

Signature  
process-sharing-unlock  sharing-lock

Arguments  
sharing-lock  A sharing lock.

Description  
The function `process-sharing-unlock` is the same as `process-unlock` but for a "sharing" lock.

Calls to `process-sharing-unlock` should be matched by calls to `process-sharing-lock`. Normally `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 181.

See also process-unlock
with-sharing-lock

**process-stop**  
*Function*

Summary  
Stops a process.
The function **process-stop** stops the process **process**.

**process** must be a full process (that is, not one created by **create-simple-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-kill**, **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 170.

**process-stop** does not return any useful value.

**process-stopped-p**  

**Function**  

**Summary**  
The predicate for stopped processes.
Package  \textit{mp}

Signature  \texttt{process-stopped-p} \texttt{process} \Rightarrow \texttt{result}

Arguments  \texttt{process} \quad \texttt{A \textit{mp}:process object.}

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

Description  The function \texttt{process-stopped-p} queries whether the process \texttt{process} is stopped or not.

If \texttt{process} stopped because it called \texttt{process-stop} on itself, then \texttt{process-stopped-p result} is \texttt{t} only if \texttt{process-stop} really stopped it (that is, a later call to \texttt{process-unstop} will unstop the process).

See also  \texttt{process-stop} \hspace{1cm} \texttt{process-unstop}

\texttt{process-unlock} \hspace{1cm} \textit{Function}

Summary  Relinquishes a lock held by the current process.

Package  \textit{mp}

Signature  \texttt{process-unlock} \texttt{lock} \&optional \texttt{errorp} \Rightarrow \texttt{result}

Arguments  \texttt{lock} \quad \texttt{The lock to be relinquished.}

\texttt{errorp} \quad \texttt{When this is \texttt{t}, an error is signalled if *current-process* is not the owner of the lock. The default is \texttt{t}.}

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

Description  Attempts to release a lock. If the lock is owned by *current-process*, \texttt{process-unlock} decrements an internal count. If
this lock count is then zero, the lock is released. Note that
\texttt{process-unlock} relates only on Lisp processes.

\textit{result} is \texttt{t} if the lock was released, and \texttt{nil} otherwise.

\textbf{Notes} \texttt{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 181.

\textbf{See also} \texttt{make-lock}
\texttt{process-exclusive-unlock}
\texttt{process-lock}
\texttt{with-lock}

\textbf{process-unstop} \hfill \textit{Function}

\textbf{Summary} Unstops a process.

\textbf{Package} \texttt{mp}

\textbf{Signature} \texttt{process-unstop process => result}

\textbf{Arguments} \textit{process} \hfill A \texttt{mp:process} object.

\textbf{Values} \textit{result} \hfill A boolean.

\textbf{Description} The function \texttt{process-unstop} unstops the process \textit{process} if it
is stopped.

\textit{process} must be a full process (that is, not one created by \texttt{create-simple-process}).

If \textit{process} was stopped (by \texttt{process-stop}), it is unstopped and
resumes execution.

\textit{result} is \texttt{t} if \textit{process} was stopped, and \texttt{nil} otherwise.
There is a discussion of a typical use of `process-unstop` in the section “Stopping and unstopping processes” on page 170.

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 `t`. This is tested periodically.

`wait-function` must not do a non-local exit. `wait-function` should not have side effects and, since it is called frequently, it should be efficient.

`wait-reason` allows you to find out why a process is waiting via the function `process-whostate`.

See also `process-wait-with-timeout`  
`process-whostate`
**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.
- *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

```plaintext
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

```plaintext
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` has the 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 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-kill`) also "poke" the process. Returning from any of the generic Process Waiting functions (see “Generic Process Wait functions” on page 183) or `cl:sleep` also implicitly pokes the process. A process may also be 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 \( t \) (because this happens at most once).

2. If there is an unhandled error in the wait function it enters the debugger like normal Lisp code, so it is easier to debug.
3. The wait function is in the dynamic scope of the calling process, and so it sees all the dynamic bindings and can throw to all the catchers. That also means that all the handlers and restarts of the calling process are applicable in the wait function.

4. The wait function can itself call Process Waiting functions or cl:sleep, with a small caveat: since these functions may implicitly "poke" the process, if the wait function calls any of them and then returns nil, it may be immediately called again (if it returns t then process-wait-local itself returns). Normally this is not a problem, because it is still waiting, but it does mean that the wait function is called more times than expected.

5. The wait function, because it can call Process Waiting functions, can use locks without causing errors. Note, however, that if the lock is held, it will cause an internal call to a Process Waiting function, which will "poke" the process and hence cause another call of the wait function (unless it returns t).

6. The wait function is visible in the output of the profiler. Another advantage of the "local" functions is that they do not interact with the scheduler and so they reduce the overhead of the scheduler.

   process-wait-local always returns t.

   See also process-wait-local-with-periodic-checks
   process-wait-local-with-timeout

**process-wait-local-with-periodic-checks**  
*Function*

**Summary**  
Like process-wait-local, but also calls the wait function periodically.

**Package**  
mp
Signature  
\texttt{process-wait-local-with-periodic-checks \textit{wait-reason} \textit{period}}  
function \texttt{&rest \textit{args}}

Arguments  
\textit{wait-reason}  
A string.  
\textit{period}  
A positive real number.  
\textit{function}  
A function designator.  
\textit{args}  
Arguments passed to \textit{function}.

Description  
The function \texttt{process-wait-local-with-periodic-checks} is like \texttt{process-wait-local}, but also calls the wait function periodically.  
The \textit{period} is in seconds.  
After each call to the function \textit{wait-function}, the process sleeps at most \textit{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 \texttt{nil}) sleeps again for at most \textit{period} seconds.

Notes  
The resolution of the period is dependent on the underlying operating system. Many systems give time-slices of few milliseconds, so the actual period may be out by a few milliseconds. In general, periods of 0.1 seconds or more are reasonably reliable, though not exact. Shorter periods become less and less reliable.  
If the period is short, the wait function is called frequently, and hence there is more overhead for the system. With a reasonable wait function and a period of 0.1 or more, this overhead is probably insignificant. If you use shorter periods, or an expensive wait function, you may want to check what the overhead is. The easiest way to check is to make sure your system is such that the wait function returns \texttt{nil}, then run...
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 \texttt{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 \texttt{process-wait-local} \br \texttt{process-wait-local-with-timeout-and-periodic-checks}

\textbf{process-wait-local-with-timeout} \hfill \textit{Function}

\textbf{Summary} \hfill Has the same semantics as \texttt{process-wait-with-timeout}, but does not interact with the scheduler.

\textbf{Package} \hfill mp

\textbf{Signature} \hfill process-wait-local-with-timeout \texttt{wait-reason \ timeout \ function \ \&rest \ args => \ result}

\textbf{Arguments} \hfill \begin{align*}
\texttt{wait-reason} & : \text{A string.} \\
\texttt{timeout} & : \text{A non-negative number.} \\
\texttt{function} & : \text{A function designator.} \\
\texttt{args} & : \text{Arguments passed to function.}
\end{align*}

\textbf{Values} \hfill \begin{align*}
\texttt{result} & : \text{A boolean.}
\end{align*}
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-wait-local`

---

**process-wait-local-with-timeout-and-periodic-checks**

*Function*

**Summary**

Like `process-wait-local-with-timeout`, but also calls the wait function periodically.

**Package**

`mp`

**Signature**

`process-wait-local-with-timeout-and-periodic-checks`

`wait-reason` `timeout` `period` `function` `&rest args`

**Arguments**

- `wait-reason` A string.
- `timeout` A non-negative number.
- `period` A positive real number.
- `function` A function designator.
- `args` Arguments passed to `function`. 
The function `process-wait-local-with-timeout-and-periodic-checks` is like `process-wait-local-with-timeout`, but also calls the wait function periodically.

The `timeout` and `period` are both in seconds.

For information about the periodic calls, see `process-wait-local-with-periodic-checks`.

See also
- `process-wait-local-with-periodic-checks`
- `process-wait-local-with-timeout`

### process-wait-with-timeout

**Function**

**Summary**
Suspend the current process until certain conditions are true, or until a timeout expires.

**Package**
mp

**Signature**

```
process-wait-with-timeout wait-reason timeout &optional
  wait-function &rest wait-arguments => bool
```

**Arguments**

- `wait-reason` A string describing the reason that the process is waiting.
- `timeout` A 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 `t`, or until `timeout` seconds have passed.

`bool` is `nil` if the timeout occurred before `wait-function` returned `true`. `bool` is true otherwise.
See also  
  process-join  
  process-wait

**process-whostate**  

*Function*

**Summary**  
Returns the state of a process.

**Signature**  
`process-whostate process => result`

**Package**  
mp

**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` 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
### pushnew-to-process-private-property

**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

**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 `#'eq`.

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.

See also

- `process-property`
- `remove-process-property`

---

**ps**

Function

Summary

Prints the processes in the system

Package

`mp`

Signature

`ps =>`

Arguments

None.
Values
None.

Description
Prints a list of the processes in the system, ordered by priority. (This function is analogous to the UNIX command `ps`.)

remove-from-process-private-property

Function

Summary
Removes a value from a private property of the current process.

Package
mp

Signature
remove-from-process-private-property indicator value &key test => result

Arguments
indicator A Lisp object.
value A Lisp object.

Values
result A list.

Description
The function `remove-from-process-private-property` removes value from the value of the private property associated with indicator for the current process.

It behaves just like `remove-from-process-property`.

See also
process-private-property
remove-from-process-property
remove-process-private-property
get-process-private-property

remove-from-process-property

Function

Summary
Removes a value from a general property of a process.
Package    mp

Signature    remove-from-process-property indicator value &key process
test => result

Arguments    indicator    A Lisp object.
value        A Lisp object.
process      A process, or nil.
test         A function designator for a function of two arguments.

Values       result       A list.

Description   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.

See also    process-property
remove-process-property

remove-process-private-property    Function

Summary    Removes a property from the private properties of the current process.
### remove-process-private-property

**Summary**
Removes a general property from a process.

**Package**
mp

**Signature**
`remove-process-private-property indicator -> removedp`

**Arguments**
- `indicator`: A Lisp object.
- `process`: A process.

**Values**
- `removedp`: A generalized boolean.

**Description**
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`
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.

See also

- `pushnew-to-process-property`
- `remove-from-process-property`
- `process-property`

### schedule-timer

**Function**

**Summary**
Schedules a timer to expire at a given time after the start of the program.

**Signature**

```
schedule-timer timer absolute-expiration-time &optional repeat-time => timer
```

**Package**

`mp`

**Arguments**

- `timer` A timer
- `absolute-expiration-time` A non-negative real
- `repeat-time` A non-negative real

**Values**

- `timer` A timer

**Description**
The `schedule-timer` function schedules a timer to expire at a given time after the start of the program. The `timer` argument is a timer, returned by `make-timer` or `make-named-timer`. The
**absolute-expiration-time** argument is a non-negative real number of seconds since the start of the program at which the timer is to expire. If **repeat-time** is specified, it is a non-negative real number of seconds that specifies a repeat interval. Each time the timer expires, it is rescheduled to expire after this repeat interval.

If the timer is already scheduled to expire at the time this function is called, it is rescheduled to expire at the time specified by the **absolute-expiration-time** argument. If that argument is nil, the timer is not rescheduled, but the repeat interval is set to the interval specified by the **repeat-time** argument.

The function **schedule-timer-relative** schedules a timer to expire at a time relative to the call to that function.

**Example**

The following example schedules a timer to expire 15 minutes after the start of the program and every 5 minutes thereafter.

```
(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
**schedule-timer-milliseconds**  

*Function*

**Summary**  
Schedules a timer to expire after a given amount of time.

**Signature**  

```
schedule-timer-milliseconds timer absolute-expiration-time  
&optional repeat-time  => timer
```

**Package**  
mp

**Arguments**

- **timer**  
A timer

- **absolute-expiration-time**  
A non-negative real

- **repeat-time**  
A non-negative real

**Values**

- **timer**  
A timer

**Description**

The `schedule-timer-milliseconds` function 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.
Example

The following example schedules a timer to expire 15 minutes after the start of the program and every 5 minutes thereafter.

```lisp
(setq timer
    (mp:make-timer 'print 10 *standard-output*))
#<Time Event : PRINT>

(mp:schedule-timer-milliseconds timer 900000 300000)
#<Time Event : PRINT>
```

See also

make-named-timer
make-timer
schedule-timer
schedule-timer-relative
schedule-timer-relative-milliseconds
timer-expired-p
timer-name
unschedule-timer

**schedule-timer-relative**

*Function*

**Summary**

Schedules a timer to expire at a given time after this function is called.

**Signature**

```lisp
schedule-timer-relative timer relative-expiration-time
&optional repeat-time => timer
```

**Package**

`mp`

**Arguments**

`timer` A timer

`relative-expiration-time` A non-negative real

`repeat-time` A non-negative real

**Values**

`timer` A timer
The `schedule-timer-relative` function 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.

**Signature**  

\[
\text{schedule-timer-relative-milliseconds timer relative-expiration-time optional repeat-time => timer}
\]

**Package**  

mp

**Arguments**  

- **timer**  A timer
- **relative-expiration-time**  A non-negative real
- **repeat-time**  A non-negative real

**Values**  

- **timer**  A timer

**Description**  The `schedule-timer-relative-milliseconds` function 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-relative-milliseconds` schedules a timer to expire at a time relative to the start of the program.
Example

The following example schedules a timer to expire 5 seconds after the call to `schedule-timer-relative-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`
`schedule-timer-milliseconds`
`schedule-timer-relative`
`timer-expired-p`
`timer-name`
`unschedule-timer`

**semaphore-acquire**

Function

**Summary**

Acquires units from a semaphore.

**Package**

`mp`

**Signature**

`semaphore-acquire sem &key timeout wait-reason count => flag`

**Arguments**

- `sem` A semaphore.
- `timeout` An integer or `nil`.
- `wait-reason` A string or `nil`.
- `count` A non-negative fixnum.

**Values**

- `flag` A generalized boolean.
Description

The function `semaphore-acquire` acquires `count` units from the semaphore `sem`.

It attempts to atomically decrement the semaphore's unit count by `count` (which defaults to 1) and returns true if this would give a non negative unit count.

If decrementing the semaphore's unit count would result in a negative number, then `semaphore-acquire` waits until the semaphore's unit count is larger than `count` and tries again. If `wait-reason` is true, then it is used as the thread's `wait-reason` when waiting for the semaphore.

If `timeout` is `nil`, `semaphore-acquire` can wait forever. 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.

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.
The function `semaphore-count` returns the current unit count of the semaphore `sem`. The value is 0 if the semaphore has no unit remaining.

The current unit count value can change in the semaphore after calling `semaphore-count`.

See also `make-semaphore`, `semaphore-acquire`, `semaphore-release`, `semaphore-wait-count`

---

**Function**

`semaphore-name`

**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`

---

**Function**

`semaphore-release`

**Summary**

Releases units back to a semaphore.

**Package**

`mp`
semaphore-release

**Signature**

`semaphore-release sem &key count => flag`

**Arguments**

- `sem` A semaphore.
- `count` A non negative fixnum.

**Values**

- `flag` A generalized boolean.

**Description**

The function `semaphore-release` releases count units back to the semaphore `sem`.

It atomically increments the semaphore's unit count by `count` (which defaults to 1).

The returned `flag` is true if any other thread was waiting for the semaphore and false otherwise.

**See also**

- `make-semaphore`
- `semaphore-acquire`
- `semaphore-count`
- `semaphore-wait-count`

---

semaphore-wait-count

**Function**

**Summary**

Get the current wait count of a semaphore.

**Package**

`mp`

**Signature**

`semaphore-wait-count sem => wait-count`

**Arguments**

- `sem` A semaphore.

**Values**

- `wait-count` A non negative fixnum.

**Description**

The function `semaphore-wait-count` returns the current number of units that other threads are waiting for from the semaphore `sem`. The value `wait-count` is 0 if the semaphore has no thread waiting for it.
The value can change in the semaphore after calling `semaphore-wait-count`.

See also `make-semaphore`, `semaphore-acquire`, `semaphore-count`, `semaphore-release`

**simple-process-p**

*Function*

Summary: A predicate identifying simple processes.

Package: `mp`

Signature: `simple-process-p object => bool`

Arguments: `object` An object

Values: `bool` A generalized boolean

Description: The `simple-process-p` function returns `t` if `object` is a simple process and `nil` otherwise.

See also `create-simple-process`

**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`
Arguments

symbol    A symbol
process   A process

Values

value    A Lisp object
flag     One of t, nil or the keyword :unbound

Description

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

(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.

Signature

timer-expired-p timer &optional delta => bool

Package

mp
Arguments

* timer  A timer
* delta  A non-negative real

Values

* bool  A boolean

Description

The `timer-expired-p` function 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.

Signature

```
timer-name timer => name
```
The \texttt{timer-name} function returns the name of the specified \texttt{timer}. The \texttt{timer} argument is a timer returned by \texttt{make-timer} or \texttt{make-named-timer}. If the timer has no name, \texttt{timer-name} returns \texttt{nil}.

The name of a timer created by either \texttt{make-timer} or \texttt{make-named-timer} can be set by means of the following syntax:

\begin{verbatim}
(setf (mp:timer-name timer) name)
\end{verbatim}

\textbf{Example}

\begin{verbatim}
(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
\end{verbatim}

\textbf{See also}

\begin{itemize}
  \item \texttt{make-named-timer}
  \item \texttt{make-timer}
  \item \texttt{schedule-timer}
  \item \texttt{schedule-timer-milliseconds}
  \item \texttt{schedule-timer-relative}
  \item \texttt{timer-expired-p}
  \item \texttt{unschedule-timer}
\end{itemize}
**unnotice-fd**

*Function*

**Summary**
Removes a file descriptor from the set of interesting input file descriptors.

**Package**
mp

**Signature**
unnotice-fd fd

**Arguments**
*fd*  A file descriptor

**Values**
None.

**Description**
The *unnotice-fd* function removes *fd* from the set of fds that cause LispWorks to wake up when they contain input.

This function is not implemented on Microsoft Windows.

**See also**
notice-fd

---

**unschedule-timer**

*Function*

**Summary**
Unschedules a scheduled timer

**Signature**
unschedule-timer timer => result

**Package**
mp

**Arguments**
*timer*  A timer

**Values**
*result*  A timer or nil

**Description**
If the specified timer has been scheduled to expire at a time after the call to *unschedule-timer*, this function unschedules the timer and returns the timer. Otherwise, the function returns nil.
The argument is a timer, returned by `make-timer` or `make-named-timer`.

Example

```lisp
(setq timer
  (mp:make-timer 'print 10 *standard-output*))

#<Time Event : PRINT>

(mp:schedule-timer-relative timer 60)

#<Time Event : PRINT>

(mp:unschedule-timer timer)

#<Time Event : PRINT>

(mp:timer-expired-p timer)

T
```

See also

- `make-named-timer`
- `make-timer`
- `schedule-timer`
- `schedule-timer-milliseconds`
- `schedule-timer-relative`
- `timer-expired-p`
- `timer-name`

---

**wait-processing-events**

**Function**

**Summary**

Waits processing events.

**Signature**

```
wait-processing-events timeout &key wait-reason wait-function wait-args => result
```

**Package**

`mp`

**Arguments**

- `timeout` A number.
- `wait-reason` A string.
- `wait-function` A function designator.
The function `wait-processing-events` does not return until one of two conditions is met:

- `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`
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
The MP Package

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.

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.

Example

To ensure that the seconds and milliseconds slots are always consistent, you can use mp:without-interrupts within the function which sets them.

```
(defstruct elapsed-time
  seconds
  milliseconds)

(defun update-elapsed-time-atomically
  (elapsed-time seconds milliseconds)
  (mp:without-interrupts
    (setf (elapsed-time-seconds elapsed-time) seconds
          (elapsed-time-milliseconds elapsed-time) milliseconds))
```
without-preemption

Macro

Summary
Identifies forms which should not be preempted during execution.

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.

yield

Function

Summary
Allows preemption to happen in low safety code.

Package
mp

Signature
yield

Arguments
None.

Values
None.

Description
Normally code compiled at safety 0 cannot be preempted because the necessary checks are omitted. This can be overcome by calling 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, \texttt{yield} can be useful. Note that \texttt{process-allow-scheduling} also allows preemption, but also checks the wait functions of other processes.

\textbf{See also} \texttt{process-allow-scheduling}
This chapter describes symbols available in the PARSERGEN package, the Lisp-Works parser generator.

This functionality is discussed in detail in Chapter 17, “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.

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>parsing-function</strong></td>
<td>The symbol name of the parsing function.</td>
</tr>
</tbody>
</table>

**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 17, “The Parser Generator”.

For a full description and examples, see Chapter 17, “The Parser Generator”. 

This chapter describes the symbols available in the SERIAL-PORT package. The Serial Port functionality is loaded into LispWorks by evaluating

```
(require "serial-port")
```

**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 SERIAL-PORT Package

This chapter applies only to LispWorks for Windows

```
37 The SERIAL-PORT Package

args
See in the Description below for details of the remaining arguments.

Values

serial-port

A serial-port object.

Description

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 maxi-
```
This chapter applies only to LispWorks for Windows

Minimum time to wait when transmitting a sequence of characters. For both reading and writing the timeout is given by the expression:

\[
\text{base\_timeout} + \text{nchars} \times \text{byte\_timeout}
\]

The default value of each of \textit{read-total-base-timeout}, \textit{read-total-byte-timeout}, \textit{write-total-base-timeout} and \textit{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

\begin{description}
\item[close-serial-port]
\item[set-serial-port-state]
\end{description}

\section*{close-serial-port \hspace{1cm} Function}

\begin{description}
\item[Summary] Closes a serial port
\item[Package] \texttt{serial-port}
\item[Signature] \texttt{close-serial-port serial-port}
\item[Arguments] \texttt{serial-port} A serial-port object.
\item[Description] The function \texttt{close-serial-port} closes the serial port associated with the given serial-port object.
\end{description}

If \texttt{serial-port} is already closed, an error is signalled.

See also

\begin{description}
\item[open-serial-port]
\end{description}

\section*{get-serial-port-state \hspace{1cm} Function}

\begin{description}
\item[Summary] Queries various aspects of the state of a serial port.
\end{description}
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
This chapter applies only to LispWorks for Windows

**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 signalled, 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 signalled.

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`

\[ \text{serial-port-input-available-p} \]

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.

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  

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  

Summary: Waits for some aspect of the state of a serial port to change.

Package: serial-port

Signature: wait-serial-port-state serial-port keys &key timeout => result

Arguments:
- serial-port: A serial-port object.
- keys: A list of keywords.
- timeout: A number.
Values

- **result**
  - A list.

Description

The function **wait-serial-port-state** waits for some state in the serial port associated with **serial-port** to change.

The argument **keys** should be a list of one or more of the keywords: :cts, :dsr, :err, :ring, :rlsd 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**

```lisp
write-serial-port-char char serial-port &optional timeout-error-p => char
```

**Arguments**

- **char**
  - A character.
- **serial-port**
  - A **serial-port** object.
- **timeout-error-p**
  - A boolean.

**Values**

- **char**
  - A character.

**Description**

The function **write-serial-port-char** writes the character **char** to the serial port associated with **serial-port**, and returns **char**.

A timeout will occur if the character cannot be written before the write timeout (as specified by values given when the serial port was opened by **open-serial-port**).
When a timeout occurs, if `timeout-error-p` is non-nil, then an error of type `serial-port-timeout` is signalled, otherwise `nil` is returned. The default value of `timeout-error-p` is `t`.

See also

**write-serial-port-string**  
*Function*  

**Summary**  
Writes a string to a serial port.

**Package**  
`serial-port`

**Signature**  

```
write-serial-port-string string serial-port &optional timeout-error-p &key start end => nwritten
```

**Arguments**  

- `string`  
  A string.
- `serial-port`  
  A `serial-port` object.
- `timeout-error-p`  
  A boolean.
- `start, end`  
  Bounding index designators for `string`.

**Values**  

- `result`  
  The string `string` or `nil`.

**Description**  
The function `write-serial-port-string` writes characters from the subsequence of `string` bounded by `start` and `end` to the serial port associated with `serial-port`.

The default values of `start` and `end` are 0 and `nil` (interpreted as the length of `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`).

---

*This chapter applies only to LispWorks for Windows*
When a timeout occurs, if `timeout-error-p` is non-nil, then an error of type `serial-port-timeout` is signalled, 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 19, “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
Add 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.
database A database.

Values

added-stream The argument stream.

Description

The add-sql-stream function 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

```lisp
(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 commit function 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.
The SQL Package

This chapter applies to the Enterprise Edition only

(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**

*database* A database.

**Description**

The `connect` function 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 221 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 the documentation for each database. General 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 223 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 226 for details.

The argument if-exists modifies the behavior of connect as follows:

: new Makes a new connection even if connections to the same database already exist.

: warn-new Makes a new connection but warns about existing connections.

: error Makes a new connection but signals an error for existing connections.

: warn-old Selects old connection if one exists (and warns) or makes a new one.
This chapter applies to the Enterprise Edition only

:old

Selects old connection if one exists or makes a new one.

The default value of if-exists is the value of *connect-if-exists*.

interface is used if connect needs to display a dialog to ask the user for username and password. If interface is a CAPI element, this is used. If interface is any other value (the default value is nil), and connect is called in a process which is associated with a CAPI interface, then this CAPI interface is used. interface has been added because dialogs asking for passwords can fail otherwise. This depends on the driver that the datasource uses: the problem has only been observed using MS SQL on Microsoft Windows.

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.

Note: 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 ...)  

encoding specifies the encoding to use in the connection. The value should be a keyword naming an acceptable encoding, or nil (the default). The value :unicode is accepted for all database-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:
If `encoding` is `nil` or `:default` then the encoding is chosen according to the default character set of the connection (if available) and if that fails the encoding `:utf-8` is used. The other recognised values of `encoding` are `:unicode`, `:utf-8`, `:ascii`, `:latin-1`, `:euc` and `:sjis`. `:unicode` uses `:utf-8` internally.

`:postgresql` encoding is ignored.

`:oracle` The only recognised values of `encoding` are `nil` and `:unicode`.

`:oracle8` encoding is ignored.

`:odbc` encoding is ignored.

`:odbc-driver` encoding is ignored.

`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-type` other than `:mysql` `signal-rollback-errors` is ignored and such an error is always signalled. 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 signalled. Otherwise rollback errors are not signalled.

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 signalled an error.
This chapter applies to the Enterprise Edition only

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.
The SQL Package

This chapter applies to the Enterprise Edition only

**Compatibility Note**

LispWorks 4.4 (and previous versions) use `connection-spec` passed to `connect` as the database name. `connect` checks if 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.

```
(connect "info")
```

The next example connects to the ODBC database `personnel` using the username "admin" and the password "secret".

```
(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:

```
(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
This chapter applies to the Enterprise Edition only

**connect-if-exists**

Variable

Summary

The default value for the *if-exists* keyword of the `connect` function.

Package

sql

Initial Value

: error

Description

The variable *connect-if-exists* is the default value for the *if-exists* keyword of the `connect` function. It can take the following values:

- :new
  Instructs `connect` to make a new connection even if connections to the same database already exist.

- :warn-new
  Instructs `connect` to make a new connection but warn about existing connections.

- :error
  Instructs `connect` to make a new connection but signal an error for existing connections.

- :warn-old
  Instructs `connect` to select an old connection if one exists (and warns) or make a new one.

- :old
  Instructs `connect` to select an old connection if one exists or make a new one.

See also

`connect`

**connected-databases**

Function

Summary

Returns a list of connected databases.

Package

sql

Signature

`connected-databases => database-list`
Arguments

None.

Values
database-list  A list of connected databases.

Description

The function connected-databases returns a list of the databases LispWorks is connected to.

See also

connect
disconnect
status
find-database
database-name

create-index  

Function

Summary

Creates an index for a table.

Package

sql

Signature

create-index name &key on unique attributes database =>

Arguments

name  The name of the index.
on  The name of a table.
unique  A boolean.
attributes  A list of attributes.
database  A database.

Values

None.

Description

The function create-index creates an index called name on the table specified by on. The attributes of the table to index are given by attributes. Setting unique to t includes UNIQUE in the SQL index command, specifying that the columns indexed must contain unique values.
The default value of unique is nil. The default value of database is *default-database*.

Example

```lisp
(create-index [manager]
  :on [emp] :unique t :attributes '((ename) [sal]))
```

See also

*default-database*
drop-index
create-table

**create-table**

*Function*

**Summary**

Creates a table.

**Package**

sql

**Signature**

`create-table name description &key database type extra-options`

**Arguments**

- `name` The name of the table.
- `description` The table properties.
- `database` A database.
- `type` A string or the keyword :support-transactions, or nil.
- `extra-options` A string or nil.

**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*.

925
type and extra-options are treated in a database-type specific way. Currently only database-type :mysql uses these options, as follows.

If type is not supplied, it defaults to the value (if any) of default-table-type that was supplied to connect. If extra-options is not supplied, it defaults to the value (if any) of default-table-extra-options that was supplied to connect.

Type, if non-nil, is used as argument to TYPE in the SQL statement:

```
create table MyTable (column-specs) TYPE = type
```

except that if type is :support-transactions then create-table will attempt to make tables that support transactions, by using the type innodb.

extra-options (if non-nil) is appended in the end of this SQL statement.

When database-type is not :mysql, type and extra-options are ignored.

Example

The following code:

```
(create-table [manager]
  '(((id) (char 10) not-null)
   ((salary) (number 8 2))))
```

is equivalent to the following SQL:

```
CREATE TABLE MANAGER
  (ID CHAR(10) NOT NULL, SALARY NUMBER(8,2))
```

See also

connect
*default-database*
drop-table

**create-view**

**Summary** Creates a view using a specified query.
This chapter applies to the Enterprise Edition only

Package  sql

Signature  

create-view name &key as column-list with-check-option
database =>

Arguments  

name  The view to be created.
as  An SQL query statement.
column-list  A list.

with-check-option  A boolean.
database  A database.

Values  None.

Description  The create-view function 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.

(create-view [manager] :as [select [*]
 :from [emp]
 :where [= dept 50]])

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

**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`. 
This chapter applies to the Enterprise Edition only

See also connect
disconnect
connected-databases
find-database
status

*default-database*

Variable

Summary The default database in database operations.

Package sql

Initial Value nil

Description The variable *default-database* is set by connect and specifies the default database to be used for database operations.

See also connect

*default-database-type*

Variable

Summary Specifies the default type of database.

Package sql

Initial Value nil

Description The variable *default-database-type* specifies the default type of database. You can set this or it is initialized by the initialize-database-type function.

LispWorks supports the values shown in “Supported databases” on page 221.
See also  
initialize-database-type

*default-update-objects-max-len*  
\begin{itemize}
\item \textbf{Summary}: The default maximum number of objects supplying data for a query when updating remote joins.
\item \textbf{Package}: sql
\item \textbf{Initial Value}: nil
\item \textbf{Description}: The variable *default-update-objects-max-len* provides the default value of the \textit{max-len} argument in the function \texttt{update-objects-joins}.
\item \textbf{See also}: update-objects-joins
\end{itemize}

\texttt{def-view-class}  
\begin{itemize}
\item \textbf{Summary}: Extends the syntax of \texttt{defclass} to allow specified slots to be mapped onto the attributes of database views.
\item \textbf{Package}: sql
\item \textbf{Signature}: \texttt{def-view-class name superclasses slots \&rest class-options => class}
\item \textbf{Arguments}:
\begin{itemize}
\item \texttt{name} A class name.
\item \texttt{superclasses} The superclasses of the class to be created.
\item \texttt{slots} The slot definitions of the new class.
\item \texttt{class-options} The class options of the new class.
\end{itemize}
\item \textbf{Values}:
\begin{itemize}
\item \texttt{class} The defined class.
\end{itemize}
\end{itemize}
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
This chapter applies to the Enterprise Edition only

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)
```
float       FLOAT
(float n)   FLOAT(n)
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.
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)
        :foreign-key
        (schema-name class-name slot-name)
        :join-class flex-property
        :retrieval :deferred))
   (:base-table flex_slot))
)
This chapter applies to the Enterprise Edition only

(def-view-class flex-property ()
  ((schema-name :type (string 8) :db-kind :key
    :column schema_name)
   (class-name :type (string 32) :db-kind :key
    :column class_name)
   (slot-name :type (string 32) :db-kind :key
    :column slot_name)
   (property :type (string 32) :db-kind :key
    :column property)
   (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

deleate-instance-records

Generic Function

Summary
Deletes records corresponding to View Class instances.

Package
sql
The SQL Package

This chapter applies to the Enterprise Edition only

### Signature
```
delete-instance-records instance =>
```

### Arguments
- `instance`: An instance of a View Class.

### Values
None.

### Description
The `delete-instance-records` function 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**

### Signature
```
delete-records &key from where database =>
```

### Arguments
- `from`: A database table.
- `where`: An SQL conditional statement.
- `database`: A database.

### Values
None.

### Description
The `delete-records` function 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*`. 
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
### disable-sql-reader-syntax

**Function**

**Summary**
Turns off square bracket syntax.

**Package**
sql

**Signature**
`disable-sql-reader-syntax` =>

**Arguments**
None.

**Values**
None.

**Description**
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

**Function**

**Summary**
Closes a connection to a database.

**Package**
sql

**Signature**
`disconnect &key database error => success`

**Arguments**
- `database` A database.
- `error` A boolean.
This chapter applies to the Enterprise Edition only

<table>
<thead>
<tr>
<th>Values</th>
<th>success</th>
<th>A boolean.</th>
</tr>
</thead>
</table>

**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`.

**Example**

```lisp
(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**

```lisp
(do-query ((&rest args) query &key database not-inside-transaction get-all) &body body =>
```

**Arguments**

- `args` A set of variables.
- `query` A database query.
The SQL Package

This chapter applies to the Enterprise Edition only

*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 255 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*, *query*, *select*, *simple-do-query*

**drop-index**

Function

Summary Deletes an index from a database.
This chapter applies to the Enterprise Edition only

**drop-index**

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: The name of a table.
- database: A database.

Values None.

Description The function drop-table deletes table from a database.

The default value of database is *default-database*.

See also create-table
     *default-database*
### drop-view

**Function**

**Summary**
Deletes a view from a database.

**Package**
sql

**Signature**
drop-view view &key database =>

**Arguments**

- **view**: A view.
- **database**: A database.

**Values**
None.

**Description**
The function `drop-view` deletes `view` from `database`.

The default value of `database` is `*default-database*`.

**Note:** `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.
This chapter applies to the Enterprise Edition only

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**

*Function*

Summary
Turns on square bracket SQL syntax.

Package
`sql`

Signature
`enable-sql-reader-syntax =>`

Arguments
None.

Values
None.

Description
The function `enable-sql-reader-syntax` turns on square bracket syntax and sets the state so that `restore-sql-reader-syntax-state` will make the syntax enabled if it is subsequently disabled.

See also
`disable-sql-reader-syntax`
`locally-disable-sql-reader-syntax`
`locally-enable-sql-reader-syntax`
`restore-sql-reader-syntax-state`
**execute-command**

Function

Summary
Executes an SQL expression.

Package
sql

Signature
execute-command sql-exp &key database =>

Arguments
sql-exp Any SQL statement other than a query.
database A database.

Values None.

Description
The function `execute-command` executes the SQL command specified by `sql-exp` for the database specified by `database`, which has a default value of `*default-database*`. The argument `sql-exp` may be any SQL statement other than a query.

To run a stored procedure, pass an appropriate string. The call to the procedure needs to be wrapped in a PL/SQL `BEGIN END` pair, for example:

```sql
(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.
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 the matching databases is returned and `count` is the number of matches. If no matching database is found and `errorp` is `nil`, then `nil` is returned. If none, or more than one, matching databases are found and `errorp` is true, then an error is signalled.

If the argument `database` is a database, it is simply returned.

See also `connect`, `connected-databases`, `database-name`, `disconnect`, `status`

### initialize-database-type

**Function**

**Summary**

Initializes a database type.

**Package**

sql

**Signature**

`initialize-database-type &key database-type => type`

**Arguments**

`database-type` A database type.

**Values**

`type` A database type.
Description

The function `initialize-database-type` initializes a database type by loading code and appropriate database libraries according to the value of `database-type`. If `*default-database-type*` is not initialized, this function initializes it. It adds `database-type` to the list of initialized types. The initialized database type is returned.

Example

The following example shows how to use `initialize-database-type` to initialize the `:odbc` database type.

```lisp
(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.
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
The SQL Package

This chapter applies to the Enterprise Edition only

(insert-records :into [person]
  :values '("abc" "Joe" "Bloggs" 10000 3000 nil "plumber")
)

is equivalent to the following SQL:

```
INSERT INTO PERSON
  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")
)
```

is equivalent to the following SQL:

```
INSERT INTO PERSON
  (PERSON_ID, INCOME, SURNAME, OCCUPATION)
  VALUES ('aaa', 10, 'jim', 'plumb')
```

The following example demonstrates how to use :av-pairs.

```
(insert-records :into [person] :av-pairs
  '(((person_id "bbb") (surname "Jones")))
)
```

See also

*default-database*
delete-records
update-records

---

**instance-refreshed**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Provides a hook for user code on View Class instance updates.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>sql</td>
</tr>
<tr>
<td>Signature</td>
<td>instance-refreshed instance</td>
</tr>
<tr>
<td>Arguments</td>
<td>instance An instance of a View Class.</td>
</tr>
</tbody>
</table>
This chapter applies to the Enterprise Edition only

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 information for the attributes in the table given by `table`.

`database` has a default value of `*default-database*`. 
If *owner* is nil, only user-owned attributes are considered. This is the default.

If *owner* is :all, all attributes are considered.

If *owner* is a string, this denotes a username and only attributes owned by *owner* are considered.

*result* is a list in which each element is a list (*attribute* *datatype* *precision* *scale* *nullable*). *attribute* is a string denoting the attribute name. *datatype* is the vendor-specific type as described in *attribute-type*. *nullable* is 1 if the attribute accepts the value NULL, and 0 otherwise.

**Example**

To print the type of every attribute in the database, do

```lisp
(loop for tab in
      (sql:list-tables)
    do
      (loop for type-info in
            (sql:list-attribute-types tab)
        do
        (format t "-&Table ~S Attribute ~S Type ~S"
                tab
                (first type-info)
                (second type-info))))
```

**See also**

- attribute-type
- list-attributes

### list-attributes

**Function**

**Summary**

Returns a list of attributes from a table in a database.

**Package**

sql

**Signature**

```lisp
list-attributes table &key database owner => result
```

**Arguments**

- `table` A table in the database.
- `database` A database.
This chapter applies to the Enterprise Edition only

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**

*Function*

**Summary**

Returns the broadcast list of streams recording SQL commands or results traffic.

**Package**

sql

**Signature**

list-sql-streams &key type database => streams

**Arguments**

type A keyword.

database A database.

**Values**

streams A list.

**Description**

The function list-sql-streams returns the broadcast list of streams recording SQL commands or results traffic.

Each element of streams is a stream or the symbol t, denoting *standard-output*.

The keyword type is one of :commands or :results, and determines whether to return a list of streams for SQL commands or results traffic.

The default value of type is :commands. The default value for database is the value of *default-database*.

**See also**

add-sql-stream
delete-sql-stream
sql-recording-p
sql-stream
This chapter applies to the Enterprise Edition only

```
start-sql-recording
stop-sql-recording
```

**list-tables**  
**Function**

**Summary** Returns a list of the table names in a database.

**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 lob-stream class 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
This chapter applies to the Enterprise Edition only

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 =>

Arguments  None.

Values  None.

Description  The function `locally-disable-sql-reader-syntax` turns off square bracket syntax and does not change syntax state. This ensures that `restore-sql-reader-syntax-state` restores the current enable/disable state.

Example  The intended use of `locally-disable-sql-reader-syntax` is in a file:

```
#.(locally-disable-sql-reader-syntax)
<Lisp code not using [...] syntax>
#.(restore-sql-reader-syntax-state)
```

See also  `disable-sql-reader-syntax`
`enable-sql-reader-syntax`
`locally-enable-sql-reader-syntax`
`restore-sql-reader-syntax-state`

locally-enable-sql-reader-syntax  Function

Summary  Turns on square bracket syntax and does not change syntax state.

Package  sql
Signature  locally-enable-sql-reader-syntax
Arguments  None.
Values  None.
Description  The function `locally-enable-sql-reader-syntax` turns on square bracket syntax and does not change the syntax state. This ensures that `restore-sql-reader-syntax-state` restores the current enable/disable state.
Example  The intended use of `locally-enable-sql-reader-syntax` is in a file:

```
#. (locally-enable-sql-reader-syntax)
<code using [...] syntax>
#. (restore-sql-reader-syntax-state)
```
See also  disable-sql-reader-syntax
enable-sql-reader-syntax
locally-disable-sql-reader-syntax
restore-sql-reader-syntax-state

**loop**

Macro

Summary  Extends `loop` to provide a clause for iterating over query results.
Package  `common-lisp`
Signature  `loop {for|as} var [type-spec] being {the|each} {records|record} {in|of} query-expression [not-inside-transaction not-inside-transaction] [get-all get-all] => result`
Arguments  `var`  A variable.
            `query-expression`  An SQL query statement.
This chapter applies to the Enterprise Edition only

**not-inside-transaction**
A generalised boolean.

**get-all**
A generalised boolean.

**Values**

**result**
A `loop` return value.

**Description**

The Common Lisp `loop` macro has been extended with a clause for iterating over query results. This extension is available only when Common SQL has been loaded. For a full description of the rest of the Common Lisp `loop` facility, including the possible return values, see the ANSI Common Lisp specification.

Each iteration of the loop assigns the next record of the table to the variable `var`. The record is represented in Lisp as a list. Destructuring can be used in `var` to bind variables to specific attributes of the records resulting from `query-expression`. In conjunction with the panoply of existing clauses available from the `loop` macro, the new iteration clause provides an integrated report generation facility.

The additional loop keywords `not-inside-transaction` and `get-all` may be useful when fetching many records through a connection with `database-type :mysql`. See the section “Special considerations for iteration functions and macros” on page 255 for details.

**Example**

This extended `loop` example performs the following on each record returned as a result of a query: bind `name` to the query result, find the salary (if any) from an associated hash-table, increment a count for salaries greater than 20000, accumulate the salary, and print the details. Finally, it prints the average salary.
(loop
  for (name) being each record in
  [select [ename] :from [emp]]
  as salary = (gethash name *salary-table*)
  initially (format t "%-20A-10D" 'name 'salary)
  when (and salary (> salary 20000))
  count salary into salaries
  and sum salary into total
  and do (format t "%-20A-10D" name salary)
  else
  do (format t "%-20A-10A" name "N/A")
  finally
  (format t "%-2&Av Salary: ~10D" (/ total salaries)))

See also
  do-query
  map-query
  query
  select
  simple-do-query

map-query

Function

Summary
Returns the results of mapping a function across an 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 An SQL query.
database A database.
not-inside-transaction A generalized boolean.
This chapter applies to the Enterprise Edition only

get-all  
A generalized boolean.

Values  
result  
A sequence of type output-type-spec containing the results of the map function.

Description  
The function map-query returns the result of mapping function across the results of query-exp. The output-type-spec argument specifies the type of the result sequence as per the Common Lisp map function.

The default value of database is *default-database*.

not-inside-transaction and get-all may be useful when fetching many records through a connection with database-type :mysql. Both of these arguments have default value nil. See the section “Special considerations for iteration functions and macros” on page 255 for details.

Example  
This example binds name to each name in the employee table and prints it.

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

See also  
do-query  
loop  
print-query  
query  
select  
simple-do-query

*mysql-library-directories*  
Variable

Package  
sql

Initial Value  
"C:\Program Files\MySQL\MySQL*\bin"
The variable *mysql-library-directories* helps Lisp-Works for Windows to locate the MySQL library for use with database-type :mysql.

It specifies a directory or a list of directories in which to search for the MySQL library. If the value is a directory pathname specifier then it is passed to directory. If the value is a list of directory pathname specifiers then each item is passed to directory. The collected results are the list of directories to search in.

The default value matches the default MySQL installation.

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-directories* such that it matches only that particular release.

See also  *mysql-library-path*

---

**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 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.
This chapter applies to the Enterprise Edition only

You can override the value of *mysql-library-path* by setting the environment variable LW_MYSQL_LIBRARY.

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.

Note: This is a direct call OCILobAppend.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.
ora-lob-assign

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.

Note: This is a direct call to OCILobAssign.

Note: this function is available only when the “oracle” module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

ora-lob-char-set-form

Summary
Returns the character set form of a LOB.

Package
sql
This chapter applies to the Enterprise Edition only

**ora-lob-char-set-form**

<table>
<thead>
<tr>
<th>Signature</th>
<th><code>ora-lob-char-set-form lob-locator &amp;key errorp =&gt; charset</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td><code>lob-locator</code> A LOB locator.</td>
</tr>
<tr>
<td></td>
<td><code>errorp</code> A generalized boolean.</td>
</tr>
<tr>
<td>Values</td>
<td><code>charset</code> A non-negative integer.</td>
</tr>
<tr>
<td>Description</td>
<td>The function <code>ora-lob-char-set-form</code> returns the char set form of the LOB underlying <code>lob-locator</code>.</td>
</tr>
<tr>
<td></td>
<td><code>charset</code> is 0 for a binary LOB (BLOB or BFILE),</td>
</tr>
<tr>
<td></td>
<td>SQLCS_IMPLICIT (1) for a character LOB (CFILE or CLOB)</td>
</tr>
<tr>
<td></td>
<td>and SQLCS_NCHAR (2) for a NCLOB.</td>
</tr>
<tr>
<td></td>
<td>If an error occurs and <code>errorp</code> is true, an error is signaled. If <code>errorp</code> is false, the function returns an object of type <code>sql-database-error</code>. The default value of <code>errorp</code> is <code>nil</code>.</td>
</tr>
<tr>
<td>Note:</td>
<td>This is a direct call to <code>OCILobCharSetForm</code>.</td>
</tr>
<tr>
<td>Note:</td>
<td>This function is available only when the &quot;oracle&quot; module is loaded. See the section “Oracle LOB interface” on page 259 for more information.</td>
</tr>
</tbody>
</table>

**ora-lob-char-set-id**

<table>
<thead>
<tr>
<th>Summary</th>
<th>Returns the database character set identifier of a LOB.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td><code>sql</code></td>
</tr>
<tr>
<td>Signature</td>
<td><code>ora-lob-char-set-id lob-locator &amp;key errorp =&gt; db-charset-id</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>lob-locator</code> A LOB locator.</td>
</tr>
<tr>
<td></td>
<td><code>errorp</code> A generalized boolean.</td>
</tr>
<tr>
<td>Values</td>
<td><code>db-charset-id</code> A non-negative number.</td>
</tr>
</tbody>
</table>
Description
The function `ora-lob-char-set-id` returns the database character set identifier of the LOB underlying `lob-locator`.

`db-charset-id` is 0 for a binary LOB.

If an error occurs and `errorp` is true, an error is signaled. If `errorp` is false, the function returns an object of type `sql-database-error`. The default value of `errorp` is `nil`.

Note: This is a direct call to OCILobCharSetID.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

---

**ora-lob-close**

**Function**

**Summary**
Closes an opened LOB.

**Package**
`sql`

**Signature**
`ora-lob-close lob-locator &key errorp`

**Arguments**

- `lob-locator` A LOB locator.
- `errorp` A generalized boolean.

**Description**
The function `ora-lob-close` closes a LOB which has been opened by `ora-lob-open`.

For more information see `ora-lob-open`.

If an error occurs and `errorp` is true, an error is signaled. If `errorp` is false, the function returns an object of type `sql-database-error`. The default value of `errorp` is `nil`.

Note: This is a direct call to OCILobClose.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.
See also  
ora-lob-open

**ora-lob-copy**  

**Function**

**Summary**
Copies part of an internal LOB.

**Package**
sql

**Signature**
ora-lob-copy dest-lob-locator src-lob-locator amount &key dest-offset src-offset errorp

**Arguments**
dest-lob-locator A LOB locator.
src-lob-locator A LOB locator.
amount A non-negative integer.
dest-offset A non-negative integer.
src-offset A non-negative integer.
errorp A generalized boolean.

**Description**
The function **ora-lob-copy** copies part of the LOB pointed to by `src-lob-locator` into the LOB pointed to by `dest-lob-locator`.

The details of the operation are determined by `amount`, `src-offset` and `dest-offset`. These numbers are in characters for CLOB/NCLOB and bytes for BLOB, and the offsets start from 1. The part of the source LOB from offset `src-offset` of length `amount` is copied into the destination LOB at offset `dest-offset`. The default value of `dest-offset` is 1 and the default value of `src-offset` is 1.

The destination LOB is extended if needed. If the `dest-offset` is beyond the end of the destination LOB, the gap between the end and `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.

**Note:** This is a direct call OCILobCopy.

**Note:** this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 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**

- db A database.
- type A Lisp object.

**Values**

- lob-locator A LOB locator.

**Description** The function ora-lob-create-empty creates an empty LOB object and returns a LOB locator for it.

If type is :lob then ora-lob-create-empty creates a LOB of type BLOB/CLOB. If type is any other value, it creates a file LOB. The default value of type is :lob.

Empty LOBs can be put in the database by passing them to insert-records or update-records. However, the preferred approach is to use the Oracle SQL function EMPTY_BLOB as described in the section “Inserting empty LOBs” on page 260.
This chapter applies to the Enterprise Edition only

The default value of \( db \) is the value of *default-database*.

**Note:** this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

---

**ora-lob-create-temporary**

**Function**

**Summary** Creates a temporary LOB.

**Package** sql

**Signature**

\[
\text{ora-lob-create-temporary} \ db-or-lob-locator \ & \text{key} \ errorp \ cache \\
session-duration \ clob-p \ => \ lob-locator
\]

**Arguments**

- **db-or-lob-locator**
  
  A database or a LOB locator.

- **errorp**
  
  A generalized boolean.

- **cache**
  
  A generalized boolean.

- **session-duration**
  
  A generalized boolean.

- **clob-p**
  
  A generalized boolean.

**Values**

- **lob-locator**
  
  A LOB locator.

**Description**

The function **ora-lob-create-temporary** creates a temporary LOB.

\( db-or-lob-locator \) specifies the database to associate the new LOB with. If it is a LOB locator the database from which the LOB locator came is used.

If an error occurs and \( errorp \) is true, an error is signaled. If \( errorp \) is false, the function returns an object of type sql-database-error. The default value of \( errorp \) is nil.

\( cache \) specifies whether to use a cache or not. The default value of \( cache \) is nil.
session-duration specifies the lifetime: if it is true then it uses OCI_DURATION_SESSION, otherwise it uses OCI_DURATION_CALL. The default value of session-duration is \texttt{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 \texttt{nil}.

The new temporary LOB locator is returned.

\textbf{Note:} This is a direct call to OCILobCreateTemporary.

\textbf{Note:} this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

See also \texttt{ora-lob-free-temporary} \texttt{ora-lob-is-temporary}

\textbf{ora-lob-disable-buffering} \hfill \textit{Function}

\textbf{Summary} \hfill Disables the buffering of the Oracle client.

\textbf{Package} \hfill sql

\textbf{Signature} \hfill \texttt{ora-lob-disable-buffering \ lob-locator \&key \ errorp}

\textbf{Arguments} \hfill

\begin{itemize}
  \item \texttt{lob-locator} \hfill A LOB locator.
  \item \texttt{errorp} \hfill A generalized boolean.
\end{itemize}

\textbf{Description} \hfill The function \texttt{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 \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}. 
Note: This is a direct call to OCILobDisableBuffering.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 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.

type A Lisp type descriptor.

**Description**  
The function **ora-lob-element-type** returns the Lisp element type that best corresponds to the charset of the LOB locator lob-locator.

For BLOB and BFILE type is (unsigned-byte 8). For CLOB, NCLOB and CFILE type is either base-char or simple-char, depending on the charset.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 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.

Note: This is a direct call to OCILobEnableBuffering.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 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 sql:p-oci-env.
This chapter applies to the Enterprise Edition only

### ora-lob-erase

**Description**

The function `ora-lob-env-handle` returns a foreign pointer to the environment handle of the LOB underlying `lob-locator`.

**Note:** this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

**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.
- `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.

Note: This is a direct call to OCILobErase.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 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.

Note: This is a direct call to OCILobFileClose.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

See also  ora-lob-file-open
This chapter applies to the Enterprise Edition only

**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` 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`.

**Note:** This is a direct call to OCILobFileCloseAll.

**Note:** this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

**See also** `ora-lob-file-close`

**ora-lob-file-exists**

*Function*

**Summary** The predicate for whether a LOB file exists.

**Package** sql

**Signature** `ora-lob-file-exists lob-locator &key errorp => result`

**Arguments**
- `lob-locator` A LOB locator.
errorp A generalized boolean.

Values  

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 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.

Note: This is a direct call to OCILobFileExists.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

ora-lob-file-get-name

Function

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).
This chapter applies to the Enterprise Edition only

If an error occurs and \textit{errorp} is true, an error is signaled. If \textit{errorp} is false, the function returns an object of type \texttt{sql-database-error}. The default value of \textit{errorp} is \texttt{nil}.

\textbf{Note:} This is a direct call to OCILobFileGetName.

\textbf{Note:} this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

\textbf{ora-lob-file-is-open} \hfill \textit{Function}

\textbf{Summary} \hfill The predicate for whether a LOB file is open.

\textbf{Package} \hfill \texttt{sql}

\textbf{Signature} \hfill \texttt{ora-lob-file-is-open lob-locator &key errorp => result}

\textbf{Arguments} \hfill
\texttt{lob-locator} \hfill A LOB locator.
\texttt{errorp} \hfill A generalized boolean.

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

\textbf{Description} \hfill The function \texttt{ora-lob-file-is-open} returns \texttt{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 \textit{errorp} is true, an error is signaled. If \textit{errorp} is false, the function returns an object of type \texttt{sql-database-error}. The default value of \textit{errorp} is \texttt{nil}.

\textbf{Note:} This is a direct call to OCILobFileIsOpen.

\textbf{Note:} this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 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.

**Note:** This is a direct call to OCILobFileOpen.

**Note:** this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 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.
This chapter applies to the Enterprise Edition only

\begin{itemize}
\item \textit{name} \quad A string or \texttt{nil}.
\item \textit{errorp} \quad A generalized boolean.
\end{itemize}

**Description**

The function \texttt{ora-lob-file-set-name} sets the directory alias and the name of the file LOB pointed to by \texttt{file-lob-locator}.

If \textit{dir-alias} is a string it should be of length no greater than 30. If it is \texttt{nil} then the directory alias of the file LOB is not changed.

If \textit{name} is a string it should be of length no greater than 255. If it is \texttt{nil} then the name of the file LOB is not changed.

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}.

**Note:** This is a direct call to OCILobFileSetAlias.

**Note:** this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

\begin{function}
\textbf{ora-lob-flush-buffer}
\end{function}

**Summary**

Flushes the buffer of the Oracle client.

**Package**

\texttt{sql}

**Signature**

\texttt{ora-lob-flush-buffer \_key free-buffer errorp}

**Arguments**

\begin{itemize}
\item \texttt{lob-locator} \quad A LOB locator.
\item \texttt{free-buffer} \quad A generalized boolean.
\item \texttt{errorp} \quad A generalized boolean.
\end{itemize}

**Description**

The function \texttt{ora-lob-flush-buffer} flushes the buffer that is used by the Oracle client.
If `free-buffer` is true, it also frees the buffer. The default value of `free-buffer` is `nil`.

If an error occurs and `errorp` is true, an error is signaled. If `errorp` is false, the function returns an object of type `sql-database-error`. The default value of `errorp` is `nil`.

**Note:** This is a direct call to OCILobFlushBuffer.

**Note:** this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

See also `ora-lob-enable-buffering`

### `ora-lob-free`

**Function**

Frees a LOB locator.

**Summary**

Frees a LOB locator.

**Package**

`sql`

**Signature**

`ora-lob-free lob-locator`

**Arguments**

`lob-locator` A LOB locator.

**Description**

The function `ora-lob-free` frees the LOB locator `lob-locator`.

If `lob-locator` was retrieved inside an iteration macro or function (that is, one of `map-query`, `do-query`, `simple-do-query` and `loop`), it is freed before the next record is fetched, or when terminating the iteration for the last record.

LOB locators which were retrieved by `select` or `query`, or were created by the user by `ora-lob-assign` or `ora-lob-create-empty` are freed automatically when the database connection is closed by a call to `disconnect`. 
If you create many LOB locators without closing the connection, it is useful to free them by calling `ora-lob-free`, to free the resources that are associated with them.

Freeing a LOB locator does not affect the underlying LOB. In particular, after modifications to the LOB there is no rollback even if there was not yet a `commit`.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

### ora-lob-free-temporary

**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`.

Note: temporary LOB locators are freed automatically when the database connection is closed by `disconnect`.

Note: This is a direct call to OCILobFreeTemporary.
Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 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

Signature
ora-lob-get-buffer lob-locator &key for-writing offset => amount/size, foreign-buffer, eof-or-error-p

Arguments
lob-locator A LOB locator.
for-writing A generalized boolean.
offset A non-negative integer or nil.

Values
amount/size A non-negative integer.
foreign-buffer A FLI pointer.
eof-or-error-p A boolean or an error object.

Description
The function ora-lob-get-buffer gets a buffer for efficient I/O with foreign functions.

If for-writing is nil, then ora-lob-get-buffer fills an internal buffer and returns three values: amount/size is how much it filled, foreign-buffer points to the actual buffer, and eof-or-error-p is the return value from the call to ora-lob-read-foreign-buffer. The offset offset is passed directly ora-lob-read-foreign-buffer.

If for-writing is true, then ora-lob-get-buffer returns two values: amount/size is the size of the foreign buffer and foreign-
buffer points to the actual buffer, which then can be passed to ora-lob-write-foreign-buffer.

The default value of for-writing is nil.

The buffer that is used by ora-lob-get-buffer is always the same for the LOB locator, it is used by ora-lob-read-buffer and ora-lob-write-buffer, and is freed automatically when the LOB locator is freed. Thus until you finish with the buffer, you cannot use ora-lob-read-buffer or ora-lob-write-buffer or call ora-lob-get-buffer again or free the LOB locator.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

Example

This first example illustrates reading using the buffer obtained by ora-lob-get-buffer. You have a foreign function:

my_chunk_processor(char *data, int size)

with this FLI definition:

(fli:define-foreign-function my_chunk_processor
 ((data :pointer)
  (size :int)))

You can pass all the data from the LOB locator to this function. Assuming no other function reads from it, it will start from the beginning.

(loop
 (multiple-value-bind (amount buffer eof-or-error-p)
    (ora-lob-get-buffer lob)
    (when (zerop amount) (return))
    (my_chunk_processor buffer amount )))

This second example illustrates writing with the buffer obtained by ora-lob-get-buffer. You have a foreign function that fills a buffer with data, and you want to write it to a
LOB. First you should lock the record, and if required trim the LOB locator.

(multiple-value-bind (size buffer)
    (ora-lob-get-buffer lob-locator
        :for-writing t
        ;; start at the beginning
        :offset 1)
    (loop (let ((amount (my-fill-buffer buffer size)))
        (when (zerop amount) (return))
        (ora-lob-write-foreign-buffer
            lob-locator nil
            amount buffer size))))

See also
ora-lob-read-buffer
ora-lob-read-foreign-buffer
ora-lob-write-buffer
ora-lob-write-foreign-buffer

**ora-lob-get-chunk-size**

*Function*

**Summary**

Returns the chunk size of a LOB.

**Package**

*sql*

**Signature**

*ora-lob-get-chunk-size* *lob-locator* &key *errorp* => *size*

**Arguments**

*lob-locator*  
A LOB locator.

*errorp*  
A generalized boolean.

**Values**

*size*  
A non-negative integer.

**Description**

The function *ora-lob-get-chunk-size* returns the chunk size of the LOB locator *lob-locator*, which is the best value for the size of a buffer.

If an error occurs and *errorp* is true, an error is signaled. If *errorp* is false, the function returns an object of type *sql-database-error*. The default value of *errorp* is *nil*.  

---
Note: This is a direct call to OCILobGetChunkSize.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

**ora-lob-get-length**

**Function**

**Summary**
Returns the length of a LOB.

**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.

Note: This is a direct call to OCILobGetLength.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

**ora-lob-internal-lob-p**

**Function**

**Summary**
The predicate for internal LOBs.

**Package**
sql
The SQL Package

This chapter applies to the Enterprise Edition only

**ora-lob-internal-lob-p**

**Function**

<table>
<thead>
<tr>
<th>Signature</th>
<th>ora-lob-internal-lob-p lob-locator =&gt; result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>lob-locator \hspace{1em} A LOB locator.</td>
</tr>
<tr>
<td>Values</td>
<td>result \hspace{1em} A boolean.</td>
</tr>
</tbody>
</table>
| Description | The function **ora-lob-internal-lob-p** returns `t` if `lob-locator` is internal (BLOB, CLOB, or NCLOB). Otherwise it returns `nil`.  
  
  **Note:** this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information. |

**ora-lob-is-equal**

**Function**

<table>
<thead>
<tr>
<th>Signature</th>
<th>ora-lob-is-equal lob-locator1 lob-locator2 =&gt; result</th>
</tr>
</thead>
</table>
| Arguments | lob-locator1 \hspace{1em} A LOB locator.  
  lob-locator2 \hspace{1em} A LOB locator. |
| Values    | result \hspace{1em} A boolean. |
| Description | The function **ora-lob-is-equal** returns `t` if `lob-locator1` and `lob-locator2` point to the same LOB object.  
  
  **Note:** This is a direct call to OCILobIsEqual.  
  
  **Note:** this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information. |
This chapter applies to the Enterprise Edition only

**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.
errorp A generalized boolean.

**Values**
result A boolean.

**Description**
The function ora-lob-is-open returns t if the LOB pointed to by lob-locator is opened (by ora-lob-open).

ora-lob-is-open is applicable to internal LOBs only.

If an error occurs and errorp is true, an error is signaled. If errorp is false, the function returns an object of type sql-database-error. The default value of errorp is nil.

**Note:** This is a direct call to OCILobIsOpen.

**Note:** this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

See also
ora-lob-open

**ora-lob-is-temporary**

Function

**Summary**
The predicate for whether a LOB is temporary.

**Package**
sql

**Signature**
ora-lob-is-temporary lob-locator &key errorp => result
Arguments

lob-locator A LOB locator.
errorp A generalized boolean.

Values

result A boolean.

Description

The function ora-lob-is-temporary returns t if the LOB underlying lob-locator is temporary, that is, it was created by ora-lob-create-temporary.

If an error occurs and 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.

Note: This is a direct call to OCILobIsTemporary.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

See also

ora-lob-create-temporary

ora-lob-load-from-file

Function

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.
This chapter applies to the Enterprise Edition only

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.

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.

Note: This is a direct call to OCILobReadFromFile. The Oracle documentation is ambiguous on whether it is mandatory to open the source LOB before calling this function.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 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 `sql:p oci-lob-locator` or `sql:p oci-file`.

**Note:** this function is available only when the “oracle” module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

---

Function: `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`.

**Note:** This is a direct call to OCILobIsInit.
Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

**ora-lob-open**

Function

Summary Opens a LOB.

Package sql

Signature ora-lob-open lob-locator &key errorp

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lob-locator</td>
<td>A LOB locator.</td>
</tr>
<tr>
<td>errorp</td>
<td>A generalized boolean.</td>
</tr>
</tbody>
</table>

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.

Note: This is a direct call to OCILobOpen.
Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 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 comparison (*unsigned-byte 8*) and *base-char* are considered as the same.

If the buffer *buffer* is not static, there is some additional overhead. For small amounts of data, this is probably insignificant.

*buffer-offset* specifies where to put the data. It is an offset in bytes from the beginning of the buffer. The default value of *buffer-offset* is 0.

*csid* specifies what Character Set ID the data in the target buffer should be. It defaults to the CSID of the LOB pointed to by *lob-locator*.

The return value *amount-read* is the number of elements (characters or bytes) that were read.

If the return value *eof-or-error-p* is *nil* then there is still more to read. If *eof-or-error-p* is *t* then it read to the end of the LOB. If an error occurred then *eof-or-error-p* is an error object.

**Note:** This is a direct call to OCILOBRead, without callback.

**Note:** this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

**Example**

This example sequentially reads the LOB data into a string, starting from offset 10000. It calls a processing function on each chunk of data and then reads in the next chunk starting from where the previous read ended.
(let ((my-buffer (make-string 1000 :element-type 'base-char))
    (offset 10000))
  (loop
    (let ((nread
      (ora-lob-read-buffer lob-locator
        offset
        1000
        my-buffer)))
      (when (zerop nread) ; end of the LOB
        (return))
      (my-processing-function my-buffer nread))
    (setq offset nil))) ; so next time it continues
                         ; from where it finished

See also
ora-lob-element-type
ora-lob-read-foreign-buffer

ora-lob-read-into-plain-file

Function

Summary
Writes the contents of a LOB into a file.

Package
sql

Signature
ora-lob-read-into-plain-file lob-locator file-name &key
offset file-offset if-exists

Arguments
lob-locator   A LOB locator.
file-name     A pathname designator.
offset        A non-negative integer, or nil.
file-offset   A non-negative integer, or nil.
if-exists     A keyword or nil.

Description
The function ora-lob-read-into-plain-file writes the contents of a LOB into a file.

description 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
This chapter applies to the Enterprise Edition only

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.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

See also ora-lob-write-from-plain-file

### *ora-lob-read-foreign-buffer* Function

<table>
<thead>
<tr>
<th>Summary</th>
<th>Reads from a LOB into a foreign buffer.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>sql</td>
</tr>
<tr>
<td>Signature</td>
<td>ora-lob-read-foreign-buffer  lob-locator offset amount foreign-buffer buffer-length &amp;key buffer-offset csid =&gt; amount-read, eof-or-error-p</td>
</tr>
<tr>
<td>Arguments</td>
<td>lob-locator A LOB locator.</td>
</tr>
<tr>
<td></td>
<td>offset A non-negative integer or nil.</td>
</tr>
<tr>
<td></td>
<td>amount A non-negative integer.</td>
</tr>
<tr>
<td></td>
<td>foreign-buffer A FLI pointer.</td>
</tr>
<tr>
<td></td>
<td>buffer-length A non-negative integer.</td>
</tr>
</tbody>
</table>
The SQL Package

This chapter applies to the Enterprise Edition only

buffer-offset  A non-negative integer.
csid  A Character Set ID

Values

amount-read  A non-negative integer.
eof-or-error-p  A boolean or an error object.

Description

The function ora-lob-read-foreign-buffer reads from the LOB pointed to by lob-locator into the foreign buffer foreign-buffer. This is just like ora-lob-read-buffer except that it reads from the LOB locator into a foreign buffer.

foreign-buffer is a FLI pointer to a buffer, which must be of size at least buffer-length.

Note: This is a direct call to OCILobRead, without callback.

Note: this function is available only when the “oracle” module is loaded. See the section “Oracle LOB interface” on page 259 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 sql:p oci-svc-ctx.
This chapter applies to the Enterprise Edition only

**Description**

The function `ora-lob-svc-ctx-handle` returns a foreign pointer to the context handle of the LOB underlying `lob-locator`.

**Note:** this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

---

**ora-lob-trim**

**Summary**

Trims an internal LOB.

**Package**

`sql`

**Signature**

`ora-lob-trim lob-locator new-size &key errorp`

**Arguments**

- `lob-locator`: A LOB locator.
- `new-size`: A non-negative integer.
- `errorp`: A generalized boolean.

**Description**

The function `ora-lob-trim` trims the LOB pointed to by `lob-locator` to a new size `new-size`, which must be smaller than its current size.

Note that `new-size` is in characters for CLOBs and bytes for BLOBs.

`ora-lob-trim` is applicable to internal LOBs only.

If an error occurs and `errorp` is true, an error is signaled. If `errorp` is false, the function returns an object of type `sql-database-error`. The default value of `errorp` is `nil`.

**Note:** This is a direct call to OCILOBTrim.

**Note:** this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.
ora-lob-write-buffer

Function

Summary
Writes a buffer to a LOB.

Package
sql

Signature
ora-lob-write-buffer lob-locator offset amount buffer &key
buffer-offset csid => amount-written, eof-or-error-p

Arguments
lob-locator A LOB locator.
offset A non-negative integer or nil.
amount A non-negative integer.
buffer A string, or a vector of element type
(unsigned-byte 8).
buffer-offset A non-negative integer.
csid A Character Set ID

Values
amount-written A non-negative integer.
eof-or-error-p A boolean or an error object.

Description
The function ora-lob-write-buffer writes to the LOB pointed to by lob-locator from buffer.

offset specifies the offset to start writing to. It starts with 1, and
specifies characters for CLOB/NCLOB/CFILE and bytes for BLOB/BFILE. If offset is nil then the offset after
the end of the previous write operation is used (read operations are
ignored). This is especially useful for writing linearly to the
LOB.

amount is the amount to write, in characters for
CLOB/NCLOB/CFILE and bytes for BLOB/BFILE.

The element type of buffer should match the element type of
the LOB locator (see ora-lob-element-type). For this com-
parison (unsigned-byte 8) and base-char are considered as
the same.
This chapter applies to the Enterprise Edition only

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.

Note: The record from which the LOB came must be locked. See the section “Locking” on page 261.

Note: This is a direct call to OCILobWrite, without callback.

Note: This function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 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
The SQL Package

This chapter applies to the Enterprise Edition only

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lob-locator</td>
<td>A LOB locator.</td>
</tr>
<tr>
<td>file-name</td>
<td>A pathname designator.</td>
</tr>
<tr>
<td>offset</td>
<td>A non-negative integer, or nil.</td>
</tr>
<tr>
<td>file-offset</td>
<td>A non-negative integer, or nil.</td>
</tr>
<tr>
<td>if-does-not-exist</td>
<td>A keyword or nil.</td>
</tr>
</tbody>
</table>

Description

The function `ora-lob-write-from-plain-file` writes the contents of a file into a LOB.

`file-name` specifies the file to read, which should be a standard file. The file is always opened in a binary mode, so if the LOB is a CLOB, the file must be in the right format when writing it into the LOB.

`offset` is the offset into the LOB from where to start writing. It starts from 1, counts characters in a CLOB, and if it is nil then the operation starts from the end of the previous write operation. The default value of `offset` is nil.

`file-offset` specifies the offset into the file to start the operation from. If `file-offset` is nil then it starts reading at the start of the file. The default value of `file-offset` is nil.

`if-does-not-exist` is passed to `open` when opening the file, with the standard Common Lisp meaning. The default value of `if-does-not-exist` is :error.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 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.
This chapter applies to the Enterprise Edition only

Package  sql

Signature  ora-lob-write-foreign-buffer lob-locator offset amount
           foreign-buffer buffer-length &key buffer-offset csid => amount-written, eof-or-error-p

Arguments  lob-locator  A LOB locator.
offset      A non-negative integer or nil.
amount      A non-negative integer.
foreign-buffer  A FLI pointer.
buffer-length  A non-negative integer.
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-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.

Note: this function is available only when the "oracle" module is loaded. See the section “Oracle LOB interface” on page 259 for more information.

See also  ora-lob-get-buffer
          ora-lob-write-buffer
### p-oci-env

**FLI type descriptor**

**Summary**
A foreign type representing objects in the Oracle interface.

**Package**
`sql`

**Description**
See “Interactions with foreign calls” on page 263 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 263 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 263 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 263 for details.
p-oci-svc-ctx

**Summary**

A foreign type representing objects in the Oracle interface.

**Package**

sql

**Description**

See “Interactions with foreign calls” on page 263 for details.

print-query

**Summary**

Prints a tabulated version of records resulting from a query.

**Package**

sql

**Signature**

```plaintext
print-query query-exp &key titles formats sizes stream database =>
```

**Arguments**

- `query-exp` An 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 `print-query` function 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 `titles`, which has a default value of `nil`.

This chapter applies to the Enterprise Edition only
The *formats* argument is a list of format strings used to print each attribute, and has a default value of `t`, which means that `-A` or `-VA` are used if sizes are provided or computed.

The field sizes are given by *sizes*. It has a default value of `t`, which specifies that minimum sizes are computed.

The output stream is given by *stream*, which has a default value of `t`. This specifies that `*standard-output*` is used.

**Examples**

The following call prints out two even columns of names and salaries:

```
(print-query [select [surname] [income] :from [person]]
  :titles '("NAME" "SALARY")')
```

See also

- `map-query`
- `print-query`
- `select`

---

**query**

*Function*

**Summary**

Queries a database and returns a list of values.

**Package**

`sql`

**Signature**

```scheme
query sql-exp &key database result-types flatp => result-list, field-names
```

**Arguments**

- `sql-exp` An SQL query statement to be performed.
- `database` A database.
- `result-types` A list of symbols.
- `flatp` A boolean.

**Values**

- `result-list` A list of values.
- `field-names` A list of strings.
This chapter applies to the Enterprise Edition only

Description
The function \texttt{query} is the basic SQL query function. It queries the database specified by \texttt{database} with an SQL query statement given by \texttt{sql-exp}.

The argument \texttt{database} defaults to *\texttt{default-database}*. 

\texttt{result-types} is a list of symbols such as \texttt{:string} and \texttt{:integer}, one for each field in the query, which are used to specify the types to return.

\texttt{flatp} is used as in \texttt{select}.

\texttt{result-list} is a list of values as per \texttt{select}, and \texttt{field-names} is a list of field names selected in \texttt{sql-exp}.

Example
The following two queries, on a table whose second column contains dates that we want to return as strings, are equivalent:

\begin{verbatim}
(sql:query "select * from some_table"
 :result-types '(nil :string))

(sql:query [select [*]
 :from [some_table]
 :result-types '(nil :string)])
\end{verbatim}

See also
\texttt{do-query}
\texttt{execute-command}
\texttt{lob-stream}
\texttt{loop}
\texttt{map-query}
\texttt{select}
\texttt{simple-do-query}

\textbf{reconnect}

\textbf{Function}

\textbf{Summary}
Reconnects a database to its underlying RDBMS.

\textbf{Package}
\texttt{sql}
The SQL Package
This chapter applies to the Enterprise Edition only

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 reconnect function reconnects database to its underlying RDBMS. If successful, success is true 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.

Note: force non-nil might result in a memory leak if the database driver fails to release its memory (some drivers do not allow the connection to be closed if the underlying RDBMS is not responding).

See also

connect
connected-databases
*default-database*
This chapter applies to the Enterprise Edition only

**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`
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.
- `all` A boolean.
- `set-operation` An SQL operation.
- `distinct` A boolean.
- `from` An SQL table.
- `result-types` A list of symbols.
- `flatp` A boolean.
- `where` An SQL condition.
- `group-by` An SQL condition.
- `having` An SQL condition.
- `database` A database.
- `order-by` An SQL condition.
This chapter applies to the Enterprise Edition only

refresh A boolean.
for-update t, :nowait, a string or a list.

Values

result-list A list of selections.

Description

The function \texttt{select} selects data from \texttt{database}, which has a default value of \texttt{*default-database*}, given the constraints specified by the rest of the arguments. It returns a list of objects as specified by \texttt{selections}. By default, the objects will each be represented as lists of attribute values.

The argument \texttt{selections} consists either of database identifiers, type-modified database identifiers or literal strings.

A type-modified database identifier is an expression such as \texttt{[foo :string]} which means that the values in column \texttt{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 253. It can also be used to return \texttt{lob-stream} objects for queries on Oracle LOB columns, using an expression like \texttt{[foo :input-stream]} or \texttt{[foo :output-stream]}

\texttt{result-types} is used when \texttt{selections} is \texttt{*} or \texttt{[*]}. It should be a list of symbols such as \texttt{:string} and \texttt{: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 \texttt{result-types} to modify the type returned from a time/date field.

The \texttt{flatp} argument, which has a default value of \texttt{nil}, specifies if full bracketed results should be returned for each matched entry. If \texttt{flatp} is \texttt{nil}, the results are returned as a list of lists. If \texttt{flatp} is \texttt{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 \texttt{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 true 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 261.

The select function 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:

\[(\text{select [person_id] [surname] :from [person]})\]

\[\Rightarrow ((111 \text{ "Brown"}) (112 \text{ "Jones"}) (113 \text{ "Smith"}))\]

In the next example, the flatp argument is set to t, and the result is a simple list of surname values:

\[(\text{select [surname] :from [person] :flatp t})\]

\[\Rightarrow (\text{"Brown" \text{ "Jones" \text{ "Smith"}}})\]

In this example data in the attribute largenum, which is of a vendor-specific large numeric type, is returned to Lisp as strings:

\[(\text{sql:select [largenum :string] :from [my-table]})\]

In this example the second column of some_table is a date that we want to return as a string:

\[(\text{sql:select [*] :from [some_table] :result-types '(nil :string)})\]

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:
The SQL Package

This chapter applies to the Enterprise Edition only

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
print-query

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.
This chapter applies to the Enterprise Edition only

Package sql

Signature

\texttt{simple-do-query (values-list query \&key names-list database not-inside-transaction get-all) \&body body =>}

Arguments

- \texttt{values-list} A variable.
- \texttt{query} A database query.
- \texttt{names-list} A variable, or \texttt{nil}.
- \texttt{database} A database.
- \texttt{not-inside-transaction} A generalized boolean.
- \texttt{get-all} A generalized boolean.
- \texttt{body} A Lisp code body.

Values None.

Description

The macro \texttt{simple-do-query} repeatedly executes \texttt{body} within a binding of \texttt{values-list} to the attributes of each record resulting from \texttt{query}.

If a variable \texttt{names-list} is supplied, then it is bound to a list of the column names for the query during the execution of \texttt{body}.

The default value of \texttt{names-list} is \texttt{nil}.

\texttt{simple-do-query} returns no values.

The default value of \texttt{database} is \texttt{*default-database*}.

\texttt{not-inside-transaction} and \texttt{get-all} may be useful when fetching many records through a connection with \texttt{database-type :mysql}. Both of these arguments have default value \texttt{nil}. See the section “Special considerations for iteration functions and macros” on page 255 for details.
Example

\begin{verbatim}
(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
\end{verbatim}

See also do-query
loop
map-query
query
select

\textbf{sql}

Function

Summary
Generates SQL from a set of expressions.

Package sql

Signature \texttt{sql \&rest args \Rightarrow sql-expression}

Arguments \texttt{args} A set of expressions.

Values \texttt{sql-expression} An SQL expression.

Description The function sql generates SQL from a set of expressions given by \texttt{args}. Each argument to sql is translated into SQL and then the \texttt{args} are concatenated with a single space between each pair. The rules for translation into SQL, based on the type of each individual argument \texttt{x}, are as follows:

\begin{itemize}
  \item \texttt{string} \Rightarrow (format nil "'-A'' x)
  \item \texttt{nil} \Rightarrow "NULL"
\end{itemize}
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-existant table.
It signifies an error that must be fixed for the code to work.
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.

Note: ODBC drivers for Oracle return the "v4 return code" as the underlying database code. Therefore in the event of an error on connection to an Oracle database, `sql-error-secondary-error-id` always returns the "v4 return code" whether the connection is through ODBC.

See also `sql-user-error`
This chapter applies to the Enterprise Edition only

**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.

Note: applicable in LispWorks for UNIX only (not LispWorks for Linux, FreeBSD, or x86/x64 Solaris).

sql-expression Function

Summary Generates an 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 An SQL expression.

Description The function **sql-expression** generates an 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*

Package  sql
Initial Value  nil

1018
This chapter applies to the Enterprise Edition only

**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.

**Note:** 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.

**Note:** applicable only on Unix.

**sql-operation**

**Function**

**Summary**

Generates an SQL statement from an operator and arguments.

**Package** `sql`

**Signature**

```
sql-operation op &rest args => sql-result
sql-operation sql-function name &rest args => sql-result
sql-operation sql-operator inop1 left &rest rights => sql-result
sql-operation sql-boolean-operator inop2 left &rest rights => sql-result
```
Arguments

- **op**: An operator.
- **args**: A set of arguments for *op*.
- **name**: An arbitrary function.
- **args**: A set of arguments for *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**: An SQL expression.

Description

The function **sql-operation** takes an operator and its arguments, and returns an SQL expression.

\[(\text{sql-operation } \text{op } \text{args})\]

is shorthand for

\[(\text{apply } (\text{sql-operator } \text{op}) \text{args}).\]

The pseudo operator **sql-function** allows an arbitrary function **name** to be passed. In this case, **name** is put in the SQL expression using **princ**, and **args** are given as arguments.

The pseudo operators **sql-boolean-operator** and **sql-operator** generate SQL that calls an infix operator with **left** on the left and **rights** on the right separated by spaces. Use **sql-boolean-operator** for SQL infix operators that return a boolean and use **sql-operator** for any other SQL infix operator.

**Note**: the pseudo operator **sql-operator** should not be confused with the Common SQL function **sql-operator**.
This chapter applies to the Enterprise Edition only

Example

The following code, uses sql-operation to produce an SQL expression.

```lisp
(sql-operation 'select
  (sql-expression :table 'foo :attribute 'bar)
  (sql-expression :attribute 'baz)
:from (list
  (sql-expression :table 'foo)
  (sql-expression :table 'quux))
:where
  (sql-operation 'or
    (sql-operation '>'
      (sql-expression :attribute 'baz)
      3)
    (sql-operation 'like
      (sql-expression :table 'foo :attribute 'bar)
      "SU%")))
```

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:

```lisp
(sql-operation 'sql-function "TO_DATE" "03/06/99"
"mm/DD/RR")
```

The following SQL expression is produced.

```
#<SQL-VALUE-EXP "TO_DATE('03/06/99','mm/DD/RR')">
```

See also

- sql
- sql-expression
- sql-operator

**sql-operator**

**Function**

**Summary**

Returns the symbol for a SQL operator.

**Package**

sql
### sql-operator

**Signature**

\[ \text{sql-operator symbol} => \text{sql-symbol} \]

**Arguments**

\text{symbol} \quad \text{A symbol naming an SQL operator.}

**Values**

\text{sql-symbol} \quad \text{A symbol.}

**Description**

The function \text{sql-operator} takes an operator as an argument and returns the Lisp symbol for the operator.

**See also**

\text{sql}

\text{sql-expression}

\text{sql-operation}

---

### sql-recording-p

**Function**

**Summary**

A predicate for determining if SQL commands or results traffic is being recorded.

**Package**

\text{sql}

**Signature**

\[ \text{sql-recording-p} \ &\text{key type database} => \text{recording-p} \]

**Arguments**

\text{type} \quad \text{One of :commands or :results.}

\text{database} \quad \text{A database.}

**Values**

\text{recording-p} \quad \text{A boolean.}

**Description**

The function \text{sql-recording-p} returns \text{t} if \text{type} is :commands and SQL commands traffic is being recorded, or if \text{type} is :results and SQL results traffic is being recorded. Otherwise it returns \text{nil}.

The default value of \text{type} is :commands. The default value of \text{database} is the value of *default-database*. 
This chapter applies to the Enterprise Edition only

See also

- add-sql-stream
- delete-sql-stream
- list-sql-streams
- sql-stream
- start-sql-recording
- stop-sql-recording

<table>
<thead>
<tr>
<th>sql-stream</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
<td>Returns the broadcast stream used for recording SQL commands or results traffic</td>
</tr>
<tr>
<td><strong>Package</strong></td>
<td>sql</td>
</tr>
<tr>
<td><strong>Signature</strong></td>
<td>sql-stream &amp;key type database =&gt; stream</td>
</tr>
</tbody>
</table>
| **Arguments** | type One of :commands or :results.  
database A database. |
| **Values** | stream A broadcast stream. |
| **Description** | The function sql-stream returns the broadcast stream used for recording SQL commands or results traffic.  
  type can be either :commands or :results, and specifies whether to return the broadcast stream for commands or results traffic.  
  The default value of type is :commands. The default value of database is the value of *default-database*.  
  Note that SQL traffic can appear on *standard-output* as well as on stream. See add-sql-stream for details. |
| **See also** | add-sql-stream  
delete-sql-stream  
list-sql-streams |
sql-temporary-error

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

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

Package sql
Superclasses simple-error
This chapter applies to the Enterprise Edition only

**sql-user-error**

*Description*
The condition class `sql-user-error` is used to signal errors in Lisp code.

*See also*
`sql-database-error`

**standard-db-object**

*Class*

*Package*
`sql`

*Superclasses*
`standard-object`

*Description*
The class `standard-db-object` implements View Classes.

*See also*
`def-view-class`

**start-sql-recording**

*Function*

*Summary*
Starts recording SQL commands or results traffic.

*Package*
`sql`

*Signature*
`start-sql-recording &key type database =>`

*Arguments*
- `type`: A keyword.
- `database`: A database.

*Values*
None.

*Description*
The function `start-sql-recording` starts recording SQL traffic, potentially to multiple streams. The traffic recorded can be the commands, the results, or both commands and results.
By default the output appears only `standard-output`. You can modify the broadcast list of recording streams using `add-sql-stream` and `delete-sql-stream`.

`type` is one of `:commands`, `:results` or `:both`. It determines whether SQL commands traffic, results traffic or both is recorded.

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`  
`list-sql-streams`  
`sql-stream`  
`sql-recording-p`  
`stop-sql-recording`

### status

#### Function

**Summary**

Returns status information for the connected databases and initialized database types.

**Package**

`sql`

**Signature**

`status &optional full =>`

**Arguments**

`full`  
A boolean.

**Values**

None.

**Description**

The function `status` prints status information to the standard output, for the connected databases and initialized database types.

If `full` is `t`, detailed status information is printed. The default value of `full` is `nil`. 
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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A keyword.</td>
</tr>
<tr>
<td>database</td>
<td>A database.</td>
</tr>
</tbody>
</table>

Values
None.

Description
The function stop-sql-recording stops recording SQL commands or results traffic.

type is one of :commands, :results or :both. It determines whether the recording of SQL commands traffic, results traffic or both is stopped.

The default value of type is :commands. The default value for database is *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 table-exists-p table &key database owner => result

Arguments table A potential table name.
database A database.
owner nil, :all or a string.

Values result A boolean.

Description The function table-exists-p determines whether there is a table named table in database database.

If owner is nil, only user-owned tables are considered. This is the default.

If owner is :all, all tables are considered.

If owner is a string, this denotes a username and only tables owned by owner are considered.

The default value of database is *default-database*.

See also list-tables

update-instance-from-records

Generic Function

Summary Updates a View Class instance.

Package sql

Signature update-instance-from-records instance &key database => instance
This chapter applies to the Enterprise Edition only

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance</td>
<td>An instance of a View Class.</td>
</tr>
<tr>
<td>database</td>
<td>A database.</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance</td>
<td>The updated View Class instance.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objects</td>
<td>A list of database objects.</td>
</tr>
<tr>
<td>slots</td>
<td>A list of slot names, or <code>t</code>.</td>
</tr>
<tr>
<td>force-p</td>
<td>A boolean.</td>
</tr>
<tr>
<td>class-name</td>
<td>The class of the objects, or <code>nil</code>.</td>
</tr>
<tr>
<td>max-len</td>
<td>A non-negative integer, or <code>nil</code>.</td>
</tr>
</tbody>
</table>
**Description**

The function `update-objects-joins` updates the remote join slots, that is those slots defined without `:retrieval :immediate`.

This is an optimization function which can improve the efficiency of an application by reducing the number of queries of the database. For each slot, it queries the database using the data from all the objects, and then assigns the appropriate value to each object.

(objects) is a list of database objects. If `class-name` is non-nil, then all the database objects are of this class. If `class-name` is `nil`, then all the database objects are of the class of the first database object in the list `objects`.

If `objects` is `nil`, then `update-objects-joins` does nothing.

`class-name` specifies a class containing all the database objects in the list `objects`. If `class-name` is `nil` (the default) then the class of the first database object is used.

(slots) provides a list of the names of slots to update. Each of these slots should be a remote join slot (as defined above).

(slots) can also be `t`, meaning update all the remote join slots. The default value of `slots` is `t`.

(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*`. 
This chapter applies to the Enterprise Edition only

See also

*default-update-objects-max-len*
def-view-class

update-records

Function

Summary
Changes the values of fields in a table.

Package
sql

Signature
update-records table &key attributes values av-pairs where database =>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>A database table.</td>
</tr>
<tr>
<td>attributes</td>
<td>A set of columns.</td>
</tr>
<tr>
<td>values</td>
<td>A set of values.</td>
</tr>
<tr>
<td>av-pairs</td>
<td>An association list alternative to attributes and values.</td>
</tr>
<tr>
<td>where</td>
<td>A condition.</td>
</tr>
<tr>
<td>database</td>
<td>A database.</td>
</tr>
</tbody>
</table>

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 doesn’t 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.
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.

`with-transaction` returns the value or multiple values returned from body.

**Example**

The following example shows how to use `with-transaction` to insert a new record, updates the department number of employees from 40 to 50, and removes employees whose sal-
ary 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.

```lisp
(with-transaction
  (insert-record :into [emp]
    :attributes '(x y z)
    :values '(a b c))
(update-records [emp]
  :attributes [dept]
  :values 50
  :where [= [dept] 40])
(delete-records :from [emp]
  :where [> [salary] 300000]))
```

**See also**

- commit
- rollback
This chapter applies to the Enterprise Edition only
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 20, “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, 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 examples/streams/buffered-stream.lisp

See also
close
stream-flush-buffer
stream-fill-buffer
stream-listen
stream-read-buffer
stream-write-buffer
with-stream-input-buffer

fundamental-binary-input-stream

Class

Summary
A stream class for binary input.

Package
stream

Superclasses
fundamental-binary-stream
fundamental-input-stream

Subclasses
None.

Description
The class fundamental-binary-input-stream provides a class for generating customized binary input stream classes. A method for stream-read-byte should be provided when using this class.
See also
fundamental-binary-stream
fundamental-input-stream
stream-read-byte

fundamental-binary-output-stream

Class

Summary
A stream class for binary output.

Package
stream

Superclasses
fundamental-binary-stream
fundamental-output-stream

Description
The class fundamental-binary-output-stream provides a class for generating customized binary output stream classes. A method for stream-write-byte should be provided.

See also
fundamental-binary-stream
fundamental-output-stream
stream-write-byte

fundamental-binary-stream

Class

Summary
A class for binary streams.

Package
stream

Superclasses
fundamental-stream

Subclasses
fundamental-binary-input-stream
fundamental-binary-output-stream

Description
The class fundamental-binary-stream is the superclass of the binary input and output stream classes. A method for stream-element-type should be provided for instantiable subclasses of this class.
fundamental-character-input-stream

Class

Summary
A class that should be included in stream classes for character input.

Package
stream

Superclasses
fundamental-character-stream
fundamental-input-stream

Subclasses
None.

Description
The class `fundamental-character-input-stream` provides default methods for generic functions used for character input, and should therefore be included by stream classes concerned with character input. The user can provide methods for these generic functions specialized on the user-defined class. Methods for other generic functions must be provided by the user.

There is an example in “Defining a new stream class” on page 270.

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**

**Class**

**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 270.

**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 `fundamental-input-stream` class is a superclass to the binary and character input classes. Its inclusion causes the generic function `input-stream-p` to return t.
See also
fundamental-binary-input-stream
fundamental-character-input-stream
fundamental-stream
input-stream-p

fundamental-output-stream

Class
Summary
A class whose inclusion causes output-stream-p to return t.
Package
stream
Superclasses
fundamental-stream
Subclasses
fundamental-binary-output-stream
fundamental-character-output-stream
Description
The fundamental-output-stream class is a superclass to the binary and character output classes. Its inclusion causes the generic function output-stream-p to return t.
See also
fundamental-binary-output-stream
fundamental-character-output-stream
fundamental-stream
input-stream-p

fundamental-stream

Class
Summary
A class whose inclusion causes stream to return t.
Package
stream
Superclasses
standard-object
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 hang-
ing.

`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 272.

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   stream       A stream.
             position-spec A file position designator.

Values      position     A file position or nil.
             success-p    A generalized boolean.

Description The generic function stream-file-position implements file-position.

stream-file-position is called when file-position is called with one argument.

(setf stream:stream-file-position) is called when file-position is called with two arguments.

The return value is returned by file-position. For the setf function, this is a slight anomaly because setf functions normally return the new value. However in this case it should return the success-p value mandated by the ANSI Common Lisp standard.

The default methods specialized on stream return nil.

stream-fill-buffer

Generic Function

Summary    Fills the stream buffer.

Package    stream
**stream-fill-buffer**

**Signature**  
\texttt{stream-fill-buffer stream \Rightarrow result}

**Arguments**  
\textit{stream}  
An input stream.

**Values**  
\textit{result}  
A generalized boolean.

**Description**  
The generic function \texttt{stream-fill-buffer} is called by the reading functions to fill an empty stream buffer from the underlying data source.

\textit{stream} should be an instance of a subclass of \texttt{buffered-stream}.

\texttt{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 \texttt{simple-string}, whose element type matches that given when the stream was constructed.

There is a built-in method specialized on \texttt{buffered-stream} which usually suffices. It calls \texttt{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 \texttt{stream-read-buffer}.

**See also**  
\texttt{buffered-stream}  
\texttt{stream-read-buffer}

---

**stream-finish-output**

**Generic Function**

**Summary**  
Implements \texttt{finish-output}.

**Package**  
\texttt{stream}

**Signature**  
\texttt{stream-finish-output stream \Rightarrow nil}
The STREAM Package

Arguments

stream  A stream.

Values

nil

Description

The generic function stream-finish-output implements finish-output. The default method is on fundamental-output-stream and does nothing.

There is an example in “Stream output” on page 272.

See also

fundamental-output-stream

stream-flush-buffer

Generic Function

Summary

Flushes a stream’s buffer.

Package

stream

Signature

stream-flush-buffer stream => result

Arguments

stream  An output stream.

Values

result  A generalized boolean.

Description

The generic function stream-flush-buffer is called by the writing functions to flush a stream buffer to the underlying data sink. stream should be an instance of a subclass of buffered-stream.

Before returning, stream-flush-buffer must set the output index of stream so that more characters can be written to the buffer. If desired, the output buffer and limit can be set too.

There is a built-in method specialized on buffered-stream which usually suffices. It calls stream-write-buffer with the
Currently active part of the stream's output buffer and sets the output index to 0.

`result` is true if the buffer was flushed.

See also

buffered-stream

stream-write-buffer

---

**stream-force-output**

*Generic Function*

**Summary**

Implements `force-output`.

**Package**

stream

**Signature**

`stream-force-output stream => nil`

**Arguments**

`stream` A stream.

**Values**

nil

**Description**

The generic function `stream-force-output` implements `force-output`. The default method is on `fundamental-output-stream` and does nothing.

There is an example in “Stream output” on page 272.

See also

`fundamental-output-stream`

---

**stream-fresh-line**

*Generic Function*

**Summary**

Used by `fresh-line` to start a new line on a given stream.

**Package**

stream

**Signature**

`stream-fresh-line stream => bool`
Arguments: stream

Values: bool

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

Values: column

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 true 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.
Description

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 :eof if end-of-file is reached, or nil if no input is available. The default method provided by fundamental-character-input-stream simply calls stream-read-char which is sufficient for file streams, but interactive streams should define their own method.

See also fundamental-character-input-stream
stream-read-char

stream-read-line

Generic Function

Summary Returns a string read from a stream.

Package stream

Signature stream-read-line stream => result terminated

Arguments stream An input stream.

Values result A string or :eof.
terminated A boolean.

Description The generic function stream-read-line reads a line of characters from stream and returns this line as a string. If the string is terminated by an end-of-file instead of a newline then terminated is t.

The default method uses repeated calls to stream-read-char, and uses stream-element-type to determine the element-type of its result.

See also fundamental-character-input-stream
stream-element-type
stream-read-char
stream-read-sequence

**Summary**
Reads a number of items from a stream into a sequence.

**Package**
*stream*

**Signature**
`stream-read-sequence stream sequence start end => index`

**Arguments**
- `stream` A stream.
- `sequence` A sequence.
- `start` An integer.
- `end` An integer.

**Values**
- `index` An integer.

**Description**
The generic function `stream-read-sequence` reads from `stream` into `sequence`. Elements from the `start` of `sequence` are replaced by elements from `stream` until `end` in `sequence` or the end-of-file in `stream` is reached. The index of the first element in `sequence` that is not replaced is returned.

A default method is provided by `fundamental-character-input-stream` which makes repeated calls to `stream-read-char` and uses `(setf elt)` to insert characters into `sequence`. A default method is provided by `fundamental-binary-input-stream` that makes repeated calls to `stream-read-byte` and also uses `(setf elt)` to insert bytes into `sequence`. Note that this may lead to error if the sequence is of inappropriate type.

**See also**
- `fundamental-binary-input-stream`
- `fundamental-character-input-stream`
- `stream-read-byte`
- `stream-read-char`
**stream-read-timeout**

*Generic Function*

Summary
Accesses the read-timeout property of a socket stream.

Package
*stream*

Signature
`stream-read-timeout stream => timeout`

Arguments
`stream` A socket stream.

Values
`timeout` A positive number or `nil`.

Description

See also
`socket-stream`
`open-tcp-stream`

**stream-start-line-p**

*Generic Function*

Summary
A generic function that returns `t` if the stream is positioned at the beginning of a line.

Package
*stream*

Signature
`stream-start-line-p stream => result`

Arguments
`stream` A stream.

Values
`result` A boolean.

Description
The generic function `stream-start-line-p` returns `t` if `stream` is positioned at the beginning of a line, and `nil` other-
wise. It is permissible to define a method that always returns nil.

Note that although a value of 0 from stream-line-column also indicates the beginning of a line, there are cases where stream-start-line-p can be meaningfully implemented and stream-line-column cannot. For example, for a window using variable-width characters the column number is not very meaningful, whereas the beginning of a line has a clear meaning.

The default method for stream-start-line-p on class fundamental-character-output-stream uses stream-line-column. Therefore, if this is defined to return nil, a method should be provided for either stream-start-line-p or stream-fresh-line.

See also fundamental-character-output-stream stream-fresh-line stream-line-column

stream-terpri

Generic Function

Summary Writes an end of line to a stream.

Package stream

Signature stream-terpri stream => nil

Arguments stream A stream.

Values nil

Description The generic function stream-terpri writes an end of line to stream, as for terpri. The default method for stream-terpri is (stream-write-char stream \Newline).
stream-unread-char

Summary
Undoes the last call to stream-read-char.

Package
stream

Signature
stream-unread-char stream character => nil

Arguments
stream A stream.
character A character.

Values
nil

Description
The generic function stream-unread-char undoes the last call to stream-read-char, as in unread-char. Every subclass of fundamental-character-input-stream must define a method for this function.

See also
fundamental-character-input-stream

stream-write-buffer

Summary
Writes a part of stream's buffer.

Package
stream

Signature
stream-write-buffer stream buffer start end

Arguments
stream An output stream.
buffer A stream buffer.
start, end Bounding indexes for a subsequence of buffer.
The generic function `stream-write-buffer` is called by `stream-flush-buffer` to write the region of the buffer bounded by `start` and `end` to the stream's underlying data sink.

`stream` should be an instance of a subclass of `buffered-stream`.

This method must be implemented for subclasses of `buffered-stream` that handle output.

See also `buffered-stream` `stream-flush-buffer`

---

**stream-write-byte**

*Generic Function*

**Summary**

A generic function used by `write-byte` to write an integer to a binary stream.

**Package**

`stream`

**Signature**

`stream-write-byte stream integer => result`

**Arguments**

`stream` A stream.

`integer` An integer.

**Values**

`result` An integer.

**Description**

The generic function `stream-write-byte` is used by `write-byte`, and writes the integer `integer` to the binary stream specified by `stream`.

A method must be implemented for all binary subclasses of `buffered-stream` that handle output. A typical implementation will convert the integer to a character using `code-char` and call `stream-write-char`. 
A method should be defined for all subclasses of \texttt{fundamental-binary-output-stream}.

See also \texttt{buffered-stream} \hfill \texttt{fundamental-binary-output-stream} \hfill \texttt{fundamental-binary-stream} \hfill \texttt{stream-write-char}

\textbf{stream-write-char} \hfill \textit{Generic Function}

Summary \quad \textit{Writes a character to a specified stream.}

Package \quad \texttt{stream}

Signature \quad \texttt{stream-write-char \ stream \ character} \Rightarrow \texttt{character}

Arguments \quad \texttt{stream} \quad \texttt{A stream.}
\quad \texttt{character} \quad \texttt{A character.}

Values \quad \texttt{character} \quad \texttt{A character.}

Description \quad The generic function \texttt{stream-write-char} \textit{writes} \texttt{character} \textit{to} \texttt{stream}. Every subclass of \texttt{fundamental-character-output-stream} must have a method defined for this function.
\quad There is an example in “Stream output” on page 272.

See also \texttt{fundamental-character-output-stream}

\textbf{stream-write-sequence} \hfill \textit{Generic Function}

Summary \quad \textit{Writes a subsequence of a sequence to a stream.}

Package \quad \texttt{stream}
Summary

Used by write-string to write a string to a character output stream.

Package

stream

Signature

stream-write-string stream string &optional start end => result

stream-write-sequence stream sequence start end => result

Arguments

stream A stream.
sequence A sequence.
start An integer.
end An integer.

Values

result A sequence.

Description

The generic function stream-write-sequence is used by
write-sequence to write a subsequence of sequence delimited
by start and end to stream.

A default method is provided by fundamental-character-
output-stream that tests each element of sequence in turn,
and then uses stream-write-char or produces an error. A
default method is provided by fundamental-binary-out-
put-stream that tests each element of sequence in turn, and
then uses stream-write-byte or produces an error.

See also

fundamental-binary-output-stream
fundamental-character-output-stream
stream-read-sequence
stream-write-byte
stream-write-char

stream-write-string

Generic Function

Summary

Used by write-string to write a string to a character output
stream.

Package

stream

Signature

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 272.

### See also

- `fundamental-character-output-stream`
- `stream-write-char`

---

### with-stream-input-buffer

**Macro**

**Summary**

Allows access to the input buffer.

**Package**

`stream`

**Signature**

```lisp
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`. 
Description

The macro with-stream-input-buffer allows access to the state of the input buffer for the given buffered stream. stream should be an instance of a subclass of buffered-stream.

Within the code body, the variables buffer, index and limit are bound to the buffer of stream, its current index and the limit of the buffer. Setting buffer, index or limit will change the values in the stream stream but note that other changes to these values (for example, by calling other stream functions) will not affect the values bound within the macro. See the example for a typical use which shows how this restriction can be handled.

The buffer is always of type simple-string. The stream-element-type of stream depends on how it was constructed.

The index is the position of the next element to be read from the buffer and the limit is the position of the element after the end of the buffer. Therefore there is no data in the buffer when index is greater than or equal to length.

Example

This example function returns a string with exactly four characters read from a buffered stream. If end-of-file is reached before four characters have been read, it returns nil.
(defun read-4-chars (stream)
  (declare (type stream:buffered-stream stream))
  (let ((res (make-string 4))
        (elt 0))
    ;; Outer loop handles buffer filling.
    (loop
      ;; Inner loop handles buffer scanning.
      (loop (stream:with-stream-input-buffer (buf ind lim) stream
            (when (>= ind lim)
              ;; End of buffer: try to refill.
              (return))
            (setf (schar res elt) (schar buf ind))
            (incf elt)
            (incf ind)
            (when (= elt 4)
              (return-from read-4-chars res)))
        (unless (stream:stream-fill-buffer stream)
              (return-from read-4-chars nil))))

See also buffered-stream
with-stream-output-buffer

with-stream-output-buffer

Macro

Summary Allows access to the output buffer.

Package stream

Signature with-stream-output-buffer (buffer index limit) stream &body body => result

Arguments buffer, index, limit

Variables

stream An output stream

body Code

Values result The value returned by body.
The macro `with-stream-output-buffer` allows access to the state of the output buffer for the given buffered stream. 

`stream` should be an instance of a subclass of `buffered-stream`.

Within the code body, the variable names `buffer`, `index` and `limit` are bound to the buffer of `stream`, its current index and the limit of the buffer. Setting `buffer`, `index` or `limit` will change the values in the stream `stream` but note that other changes to these values (for example, by calling other stream functions) will not affect the values bound within the macro. See the example for a typical use which shows how this restriction can be handled.

The buffers are always of type `simple-string`. The `stream-element-type` of `stream` depends on how the stream was constructed.

The index is the position of the next free element in the buffer and the limit is the position of the element after the end of the buffer. Therefore the buffer is full when `index` is greater than or equal to `length`.

This example function writes a four character string to a buffered stream.
(defun write-4-chars (stream string)
  (declare (type stream:buffered-stream stream))
  (let ((elt 0))
    ;; Outer loop handles buffer flushing.
    (loop
      ;; Inner loop handles buffer updating.
      (loop (stream:with-stream-output-buffer (buf ind lim) stream
        (when (>= ind lim)
          ;; Buffer full: try to flush.
          (return))
        (setf (schar buf ind) (schar string elt))
        (incf elt)
        (incf ind)
        (when (= elt 4)
          (return-from write-4-chars)))))
    (stream:stream-flush-buffer stream))))

See also buffered-stream
with-stream-input-buffer
This chapter describes symbols available in the **SYSTEM** package.
Various uses of the symbols documented here are discussed throughout this manual.

**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 :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 \textit{func}.

The values returned from the call to \textit{func} with \textit{args}.

The function \texttt{apply-with-allocation-in-gen-num} applies the function \textit{func} to \textit{args} such that objects of allocation type \textit{what} are allocated in generation \textit{gen-num}, in 64-bit LispWorks.

See also the keyword \texttt{:allocation} to \texttt{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.

Note that allocation of interned symbols is controlled separately by \texttt{*symbol-alloc-gen-num*}.

\textbf{Note:} In 32-bit LispWorks the argument \textit{what} is ignored and the effect is like that of the macro \texttt{allocation-in-gen-num}.

\texttt{atomic-decf}\hspace{1cm}\texttt{atomic-incf}\hspace{1cm}\textbf{Macros}

Like \texttt{incf} and \texttt{decf}, but does the operation atomically.

\begin{verbatim}
atomic-decf place &optional delta => new-value
atomic-incf place &optional delta => new-value
\end{verbatim}
Arguments
place One of the specific set of places defined for low level atomic operations.
delta A number, default value 1.

Values
new-value A number

Description
The macro atomic-decf is like decf and atomic-incf is like incf, except that they are guaranteed atomic for a suitable place.

place must be one of the places described in “Low level atomic operations” on page 175, or expand to one of them.

Notes
Unlike atomic-fixnum-decf and atomic-fixnum-incf, these macros can deal with any number.

See also
atomic-fixnum-decf
atomic-fixnum-incf
low-level-atomic-place-p

atomic-exchange

Macro

Summary
Atomically exchange a place value with a new value, returning the old value.

Package
system

Signature
atomic-exchange place new-value => old-value

Arguments
place One of the specific set of places defined for low level atomic operations.
new-value An object.

Values
old-value An object.
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 175, 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 `decf` and `incf`, but does the operation atomically.

**Package**

`system`

**Signature**

`atomic-fixnum-decf place &optional fixnum-delta => new-value`

`atomic-fixnum-incf place &optional fixnum-delta => new-value`

**Arguments**

`place` One of the specific set of places defined for low level atomic operations.

`fixnum-delta` A fixnum, default value 1

**Values**

`new-value` A fixnum.

**Description**

The macro `atomic-fixnum-decf` is like `decf` (for fixnums only) and `atomic-fixnum-incf` is like `incf` (for fixnums only), except that they are guaranteed atomic for a suitable `place`.

`place` must be one of the places described in “Low level atomic operations” on page 175, 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.
atomic-pop

Macro

Summary Like pop, but does the operation atomically.

Package system

Signature atomic-pop place => element

Arguments

place One of the specific set of places defined for low level atomic operations.

Values element An object.

Description The macro atomic-pop is the same as cl:pop, but is guaranteed atomic for a suitable place.

place must be one of the places described in “Low level atomic operations” on page 175, or expand to one of them.

See also atomic-push

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.

See also atomic-decf
atomic-incf
low-level-atomic-place-p
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 \texttt{atomic-push} is the same as \texttt{cl:push}, but is guaranteed atomic for a suitable place.

\texttt{place} must be one of the places described in “Low level atomic operations” on page 175, or expand to one of them.

Notes In many cases the natural inverse of \texttt{push} is \texttt{delete}, but there is no way to do \texttt{delete} atomically, except by using a separate lock, which must also be held while doing the \texttt{push}.

See also \texttt{atomic-pop} \texttt{low-level-atomic-place-p}

\texttt{augmented-string} Type

Summary The augmented string type.

Package \texttt{system}

Signature \texttt{augmented-string length}

Arguments \texttt{length} The length of the string (or *, meaning any).

Description This is the string type that can hold any character. Equivalent to:

\[(\text{vector character length})\]

\texttt{augmented-string-p} Function

Summary Tests if an object is an augmented string.
Package system

Signature `augmented-string-p object => bool`

Arguments `object` The object to be tested.

Values `bool` `t` if `object` is an augmented string; `nil` otherwise.

Description This is the predicate for augmented strings.

See also `augmented-string`

call-system

Function

Package system

Signature `call-system command &key current-directory wait shell-type => status`

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.

Description `call-system` allows executables and DOS or Unix shell commands to be called from Lisp code. The output goes to standard output, as the operating system sees it. (This normally means `*terminal-io*` in LispWorks.)
If command is a string then it is passed to the shell as the command to run without any other arguments. The type of shell to run is determined by shell-type as described below.

If command is a list then it becomes the argv of a command to run directly, without invoking a shell. The first element is the command to run directly and the other elements are passed as arguments on the command line (that is, element 0 has its name in argv[0] in C, and so on).

If command is a simple vector of strings, the element at index 0 is the command to run and the other elements are the complete set of arguments seen by the command (that is, element 1 becomes argv[0] in C, and so on).

If command is nil, then the shell is run.

On Microsoft Windows current-directory is the lpCurrentDirectory argument passed to CreateProcess. If this is not supplied, the pathname-location of the current-pathname is passed.

If wait is true, call-system does not return until the process has exited. The default for wait is t.

On 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 Microsoft Windows if shell-type is nil then cmd.exe is used on Windows Vista, Windows XP and Windows 2000 and command.com on Windows 98 and Windows ME.

call-system returns the exit status of the shell invoked to execute the command on Unix/Linux/Mac OS X, or the process created on Microsoft Windows.

Compatibility Note

The :shell-type argument is not implemented in LispWorks for Windows 4.4 and earlier, and cmd.exe is not used implicitly.
LispWorks for Windows 5.0 and later use *shell-type* cmd.exe (or command.com) by default when *command* is a string. The user may see a DOS command window in this case. To call your command directly *command* should be a list, as in the last example below.

**Example**

On Unix:

```lisp
(call-system (format nil "adb ~a < ~a > ~a"
               (namestring a)
               (namestring b)
               (namestring c)))
```

On Microsoft Windows:

```lisp
(sys:call-system "sleep 3" :wait t)
(sys:call-system '("notepad" "myfile.txt"))
```

See also

open-pipe  
call-system-showing-output  
run-shell-command

---

**call-system-showing-output**

*Function*

**Package**  
*system*

**Signature**

```lisp
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*  
A symbol.
**The SYSTEM Package**

**wait**
A boolean.

**shell-type**
A string. Supported only on Unix/Linux/Mac OS X.

**kill-process-on-abort**
A generalized boolean.

**Values**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>status</strong></td>
<td>The exit status of the invoked shell or process.</td>
</tr>
</tbody>
</table>

**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`.

`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. 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.
## The SYSTEM Package

### Package

| system |

### Signature

```
cdr-assoc item alist &key test test-not key => result
(setf cdr-assoc) value item alist => value
```

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>An object.</td>
</tr>
<tr>
<td>alist</td>
<td>An association list.</td>
</tr>
<tr>
<td>test</td>
<td>A function designator.</td>
</tr>
<tr>
<td>test-not</td>
<td>A function designator.</td>
</tr>
<tr>
<td>key</td>
<td>A function designator.</td>
</tr>
<tr>
<td>value</td>
<td>An object.</td>
</tr>
</tbody>
</table>

### Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>An object (from alist) or nil.</td>
</tr>
</tbody>
</table>

### Description

The functions `cdr-assoc` and `(setq 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 `(setq cdr-assoc)` read and write the cdr of an element in a manner consistent with the Common Lisp notion of places.

`cdr-assoc` returns the cdr of the first cons in the alist `alist` that satisfies the test, or `nil` if no element of `alist` matches.

`(setq cdr-assoc)` modifies the first cons in `alist` that satisfies the test, setting its cdr to `value`. If no element of `alist` matches, then `(setq cdr-assoc)` constructs a new cons `(cons item value)` and inserts it in the head of `alist`. 
Example

CL-USER 1 > (defvar *my-alist*
(list (cons :foo 1)
(cons :bar 2)))
*MY-ALIST*

CL-USER 2 > (setf (sys:cdr-assoc :bar
*my-alist*) 3)
3

CL-USER 3 > *my-alist*
((:FOO . 1) (:BAR . 3))

*check-network-server*  

Variable
Summary  
Indicates the presence of a network license.

Note: LispWorks for UNIX only.

Package  
system

Description  
This should always be set to t for a site (that is, network) license — the licensing mechanism does not work in any other circumstances. Do not set the variable otherwise, as it overrides any useful diagnostics which may accompany key-file errors. Not applicable to LispWorks for Linux, Windows, x86/x64 Solaris, FreeBSD 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 one of the known Gesture Spec keywords, or its char-code, and gspec’s modifiers contains its char-bits attribute mapped onto the values gesture-spec-control-bit etc.

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

(sys:coerce-to-gesture-spec #\Control-C)
=>
#S(SYSTEM::GESTURE-SPEC :DATA 67 :MODIFIERS 2)

CL-USER 8 > (sys:coerce-to-gesture-spec #\Control-\c)
=>
#S(SYSTEM::GESTURE-SPEC :DATA 99 :MODIFIERS 2)

(sys:coerce-to-gesture-spec :F10)
=>
#S(SYSTEM::GESTURE-SPEC :DATA :F10 :MODIFIERS 0)

(sys:coerce-to-gesture-spec "Ctrl-C")
=>
#S(SYSTEM::GESTURE-SPEC :DATA 67 :MODIFIERS 2)

(sys:coerce-to-gesture-spec "Shift-F10")
=>
#S(SYSTEM::GESTURE-SPEC :DATA :F10 :MODIFIERS 1)

See also
gesture-spec-control-bit
gesture-spec-data
gesture-spec-modifiers
gesture-spec-p
gesture-spec-to-character
make-gesture-spec
print-pretty-gesture-spec

compare-and-swap

Macro

Summary
Performs a conditional store, atomically.

Package
system

Signature
compare-and-swap place compare new-value => result

Arguments
place One of the specific set of places defined for low level atomic operations.
compare An object.
new-value  An object.

Values  result  A boolean.

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 175, 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 (setq 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 if it already copied to new-path in the current session, and if so does nothing.
2. It checks if the flag-name entry exists, and if so it does nothing.
3. It checks if 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 if 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.

**Note:** this function 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-snapshot* 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-snapshot* is nil LispWorks behaves like LispWorks 5.0 and previous versions. That is, it attempts to enter the command line debugger.

default-eol-style

Function

Summary
Provides a default end of line style for a file.

Package
system

Signature
default-eol-style pathname ef-spec buffer length => new-ef-spec

Arguments
pathname
Pathname identifying location of buffer.

ef-spec
An external format spec.

buffer
A buffer whose contents are examined.

length
Length (an integer) up to which buffer should be examined.

Values
new-ef-spec
A new external format spec created by merging ef-spec with the encoding that was found.

Description
Merge ef-spec with (:default :eol-style :crlf) on Microsoft Windows, (:default :eol-style :lf) on 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 (mark-and-sweep 2) 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. It 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  

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.

**Note:** 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))
  ~(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

`:apropos-noeval foo`

will find all the symbols containing the string `foo`, whereas

`(setq foo "bar")`
:apropos-eval foo

will find all the symbols containing the string bar.

---

**detect-eol-style**

*Function*

**Summary**

Detects the end of line style of a file.

**Package**

system

**Signature**

detect-eol-style pathname ef-spec buffer length => new-ef-spec

**Arguments**

- **pathname**  
  Pathname identifying location of buffer.
- **ef-spec**  
  An external format spec.
- **buffer**  
  A buffer whose contents are examined.
- **length**  
  Length (an integer) up to which buffer should be examined.

**Values**

- **new-ef-spec**  
  A new external format spec created by merging ef-spec with the encoding that was found.

**Description**

When the encoding in ef-spec has foreign type (**unsigned-byte 8**), search buffer up to length for the first occurrence of the byte (10). If found, and it is preceded in buffer by (13), merge ef-spec with

```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

(:default :sol-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

(:default :sol-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 \emph{jis}, \emph{sjis}, \emph{euc}, \emph{unicode} and \emph{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*

\textbf{detect-unicode-bom}

\textit{Function}

\textbf{Summary}
Looks for the Unicode Byte Order Mark, which if found is assumed to indicate a Unicode UCS-2 encoded file.

\textbf{Package}
\texttt{system}

\textbf{Signature}
detect-unicode-bom pathname ef-spec buffer length => new-ef-spec

\textbf{Arguments}
\begin{itemize}
  \item \texttt{pathname} \hspace{1cm} Pathname identifying location of buffer.
  \item \texttt{ef-spec} \hspace{1cm} An external format spec.
  \item \texttt{buffer} \hspace{1cm} A buffer whose contents are examined.
  \item \texttt{length} \hspace{1cm} Length (an integer) up to which buffer should be examined.
\end{itemize}

\textbf{Values}
\begin{itemize}
  \item \texttt{new-ef-spec} \hspace{1cm} A new external format spec created by merging \texttt{ef-spec} with the encoding that was found.
\end{itemize}

\textbf{Description}
When called as part of \texttt{open}'s encoding detection routine, if byte pair FE FF is found at the start of the file, it is assumed to be UTF16-BE encoded. This encoding is represented by the ef-spec \texttt{(:unicode :little-endian nil)}.

If byte pair FF FE is found at the start of the file, it is assumed to be UTF16-LE encoded. This encoding is represented by the ef-spec \texttt{(:unicode :little-endian t)}.
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 177 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**

**Summary**
Ensures all following stores and loads in the program are executed after all prior stores.

**Package**
**system**

**Function**

**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 177 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 177 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.
See “Ensuring order of memory between operations in different threads” on page 177 for a full description and example.

Notes
You should have a good understanding of multiprocessing issues at the CPU level to write code that actually needs this.

See also
ensure-loads-after-loads
ensure-memory-after-store
ensure-stores-after-memory

*extended-spaces*  
Variable

Summary
Extends the notion of space to include more than just the space character.

Package
system

Initial value
nil

Description
When this variable is true, the concept of “space” is extended from just \Space to include other appropriate characters. The default is nil, for ANS compliance, but we recommend that you set it to t.

This variable controls how the format directives ~:C and ~:@C output graphic characters which have an empty glyph. When this variable is t, all such characters are output using the name:

\begin{verbatim}
(format nil "~:C" \No-break-space)  -> "No-Break-Space"
(format nil "~:@C" (code-char #x3000))  -> "Ideographic-Space"
\end{verbatim}

When false, only one such character is output using the name:
(format nil "-:C" \Space) ➞ "Space"
(format nil "-:C" \No-break-space) ➞ " "
(format nil "-:C" (code-char #x3000)) ➞ " "

It also affects whitespace-char-p.

See also extended-character-p

*file-encoding-detection-algorithm*

Variable

Summary List of functions to call to work out an encoding.

Package system

Initial value (find-filename-pattern-encoding-match
find-encoding-option
detect-unicode-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.

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:

(find-encoding-option locale-file-encoding)

There are further examples in “Guessing the external format” on page 297.
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

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 signalled when open, load or compile-file fail to detect an external format to use.
The ef-spec slot contains the incomplete external format specification argument constructed by guess-external-format.

See also

guess-external-format

*file-eol-style-detection-algorithm*

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 find-encoding-option pathname ef-spec buffer length => result

Arguments
pathname Pathname identifying location of buffer.

ef-spec An external format spec.

buffer A buffer whose contents are examined.

length Length (an integer) up to which buffer should be examined.
Values  
result  
The result of reading the value returned from the encoding or external-format option as a Lisp expression in the keyword package.

Description  
Looks in the file options (EMACS-style -*- line) for an option called encoding or external-format, with value value.

If found, it reads value as a Lisp expression in the keyword package and merges ef-spec with value and returns the result as result. Thus it does not override a supplied ef-spec.

See also  
*file-encoding-detection-algorithm*

**find-filename-pattern-encoding-match**  
*Function*

Summary  
Finds the encoding of a file based on the filename.

Package  
system

Signature  
find-filename-pattern-encoding-match pathname ef-spec buffer length => new-ef-spec

Arguments  
pathname  
Pathname identifying location of buffer.

ef-spec  
An external format spec.

buffer  
A buffer whose contents are examined.

length  
Length (an integer) up to which buffer should be examined.

Values  
new-ef-spec  
An external format spec.

Description  
Compares pathname (using pathname-match-p) with elements of *filename-pattern-encoding-matches*.
If a match is found, merges ef-spec with the corresponding external format spec and returns the result as new-ef-spec. Thus it does not override a supplied ef-spec.

See also

*file-encoding-detection-algorithm*
*filename-pattern-encoding-matches*

---

**gen-num-segments-fragmentation-state**

**Function**

**Summary**
Shows the fragmentation state in a generation in 64-bit LispWorks.

**Package**
**system**

**Signature**

```
gen-num-segments-fragmentation-state gen-num &optional statics-too => fragmentation-state
```

**Arguments**

- **gen-num**
  A number.
- **statics-too**
  A generalized boolean?

**Values**

```
fragmentation-state
```

A list in which each element is a list of length 3.

**Description**

The function `gen-num-segments-fragmentation-state` shows the fragmentation state in a generation in 64-bit 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.

Note: The implementation of set-blocking-gen-num is intended to solve any fragmentation issues automatically.

Note: 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

---

**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 10, “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.

**gesture-spec-accelerator-bit**

*Constant*

**Summary**

Used in the representation of a keystroke with the accelerator key.

**Package**

`system`

**Description**

The constant `gesture-spec-accelerator-bit` is used to represent the accelerator key in a Gesture Spec object.


**See also**

`coerce-to-gesture-spec`
`gesture-spec-modifiers`
`make-gesture-spec`

**gesture-spec-control-bit**

*Constant*

**Summary**

Used in the representation of a keystroke with the Control key.

**Package**

`system`

**Description**

The constant `gesture-spec-control-bit` is used to represent the Control modifier key in a Gesture Spec object.

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 `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-hyper-bit**

*Constant*

**Summary**

Used in the representation of a keystroke with the `Hyper` key.

**Package**

`system`
Description

The constant `gesture-spec-hyper-bit` is used to represent the `Hyper` modifier key in a Gesture Spec object.


See also

coerce-to-gesture-spec
gesture-spec-modifiers
make-gesture-spec

gesture-spec-meta-bit

Constant

Summary

Used in the representation of a keystroke with the `Meta` key.

Package

`system`

Description

The constant `gesture-spec-meta-bit` is used to represent the `Meta` modifier key in a Gesture Spec object.


See also

coerce-to-gesture-spec
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

Function  

`gesture-spec-p`  

Summary  
The predicate for Gesture Spec objects.

Package  
`system`

Signature  
`gesture-spec-p object => result`

Arguments  
`object`  A Lisp object

Values  
`result`  A boolean.

Description  
The function `gesture-spec-p` is the predicate for whether the object `object` is a Gesture Spec object.
gesture-spec-shift-bit

**Constant**

**Summary**
Used in the representation of a keystroke with the **Shift** key.

**Package**
`system`

**Description**
The constant `gesture-spec-shift-bit` is used to represent the **Shift** modifier key in a Gesture Spec object.

Note that 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.


**See also**
- `coerce-to-gesture-spec`
- `gesture-spec-modifiers`
- `make-gesture-spec`

gesture-spec-super-bit

**Constant**

**Summary**
Used in the representation of a keystroke with the **Super** key.

**Package**
`system`

**Description**
The constant `gesture-spec-super-bit` is used to represent the **Super** modifier key in a Gesture Spec object.

See also
- `coerce-to-gesture-spec`
- `gesture-spec-modifiers`
- `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 => char`

**Arguments**
- `gspec`: A Gesture Spec object

**Values**
- `char`: A Lisp character.

**Description**
The function `gesture-spec-to-character` returns the Lisp character object corresponding to the Gesture Spec object `gspec`.

Modifier bits in `gspec` are mapped to Lisp character bits attributes where possible. `gesture-spec-accelerator-bit` is ignored.

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.
The size of the file in bytes.

The number of 512-bytes blocks used by the file.

The protection value of the file.

The time of the last access to the file in seconds from 1 January 1970.

The time of the last change in the data of the file in seconds from 1 January 1970.

The time of the last modification of the file status in seconds from 1 January 1970.

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 on a Microsoft Windows or Mac OS X machine.

**Package**

`system`
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 only on Microsoft Windows and Mac OS X.

what indicates the purpose of the special folder. For instance, :common-appdata means the folder containing application data for all users.

The following values are recognized on Microsoft Windows and Mac OS X:


:documents is an alias for :my-documents.

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. The default value of create is nil.

If the folder does exist, result is nil.
In LispWorks 5.0 and previous versions, \texttt{get-folder-path} returns a string.

This form constructs a pathname to a file \texttt{foo.lisp} in the user’s documents directory:

\begin{verbatim}
(make-pathname
 :name "foo"
 :type "lisp"
 :defaults
 (sys:get-folder-path :my-documents))
\end{verbatim}

See also \texttt{get-user-profile-directory}

\textbf{get-user-profile-directory} \hfill Function

\textbf{Summary} \hspace{1cm} Gets the root of the user’s profile on a Windows NT-based system.

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

\textbf{Signature} \hspace{1cm} \texttt{get-user-profile-directory => result}

\textbf{Values} \hspace{1cm} \texttt{result} \hspace{1.5cm} A directory pathname naming the path, or \texttt{nil}.

\textbf{Description} \hspace{1cm} 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 2000, Windows XP and Windows Vista). \texttt{get-user-profile-directory} is implemented only on Microsoft Windows.

\texttt{result} names the root of the profile directory.

Note that the default path for each user’s profile may differ between versions of the operating system.
In LispWorks 5.0 and previous versions, `get-user-profile-directory` returns a string.

Example

On Windows XP:

```lisp
(sys:get-user-profile-directory)
=>
#P"C:/Documents and Settings/fred/"
```

On Windows 98 SE:

```lisp
(sys:get-user-profile-directory)
=>
nil
```

See also `get-folder-path`

---

**guess-external-format**

*Function*

Tries to work out the external format

**Summary**

**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 signalled.
The caller offers a restart for trying again with respecified
external-format and/or element-type arguments. Other-
wise 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 exter-
nal 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 signalled.

See also  
*file-encoding-detection-algorithm*  
*file-eol-style-detection-algorithm*  
file-encoding-resolution-error

### in-static-area

**Macro**

**Summary**  
Allocates the objects produced by the specified forms to the
static area.

**Package**  
**system**

**Signature**  
in-static-area &rest body => result

**Arguments**  
`body`  
The forms for which you want the garbage
collector to allocate space in the static area.

**Values**  
`result`  
The result of executing `body`.

**Description**  
Allocates the objects produced by the specified forms to the
static area. Objects in the static area are not moved, though
they are garbage collected when there is no longer a pointer
to the object.
Note: the macro `in-static-area` is deprecated. Use `make-array` with `:allocation :static` where possible instead.

Example

```lisp
(system:in-static-area (make-string 10))
```

See also

- `enlarge-static`
- `make-array`
- `staticp`

### int32

**Type**

A type used to generate optimal 32-bit arithmetic code.

**Summary**

The type `int32` is used to generate optimal 32-bit arithmetic code.

**Package**

`system`

**Signature**

`int32`

**Description**

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 95 for more information.

**See also**

- `int32*`
- `int32+`
- `int32-`
- `+int32-0+`
- `+int32-1+`
- `int32-1+`
- `int32-1-`
- `int32/`
- `int32/=`
```
int32<
int32<=
int32=
int32>
int32>=
int32-aref
int32-logand
int32-logandc1
int32-logandc2
int32-logeqv
int32-logior
int32-lognand
int32-lognor
int32-lognot
int32-logorc1
int32-logorc2
int32-logxor
int32-minusp
int32-plusp
int32-to-integer
int32-zerop
integer-to-int32
make-simple-int32-vector
simple-int32-vector
```

### int32*

**Function**

The multiply operator for int32 objects.

**Summary**

**Package**

system

**Signature**

\[ \text{int32* } x \ y \Rightarrow \text{int32} \]

**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 $\text{int32}$ An int32 object.

Description The function int32* is the multiply operator for int32 objects.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also int32

Function

int32+

Summary The add operator for int32 objects.

Package system

Signature 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 $\text{int32}$ An int32 object.

Description The function int32+ is the add operator for int32 objects.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also int32
**Function**

**int32-**

**Summary**
The subtract operator for int32 objects.

**Package**
`system`

**Signature**
`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 subtract operator for int32 objects.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

**See also**
`int32`

---

**Symbol Macro**

**+int32-0+**

**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`
**Symbol Macro**

**+int32-1+**

**Summary**
Shorthand for *(sys:integer-to-int32 1).*

**Package**
system

**Description**
The symbol macro +int32-1+ expands to *(sys:integer-to-int32 1).*

**See also**
integer-to-int32

---

**Function**

**int32-1+**

**Summary**
The operator for int32 objects corresponding to the function 1+.

**Package**
system

**Signature**
int32-1+ x => int32

**Arguments**
x
An int32 object or an integer of type *(signed-byte 32).*

**Values**
int32
An int32 object.

**Description**
The function int32-1+ is the operator for int32 objects that corresponds to the function 1+.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

**See also**
int32
### int32-1-

**Function**

**Summary**
The operator for int32 objects corresponding to the function 1-.

**Package**
`system`

**Signature**
`int32-1- x => int32`

**Arguments**
- `x`: An int32 object or an integer of type `(signed-byte 32)`.

**Values**
- `int32`: An int32 object.

**Description**
The function `int32-1-` is the operator for int32 objects that corresponds to the function 1-.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also: int32

### int32/

**Function**

**Summary**
The divide operator for int32 objects.

**Package**
`system`

**Signature**
`int32/ x y => int32`

**Arguments**
- `x`: An int32 object or an integer of type `(signed-byte 32)`.
- `y`: An int32 object that does not correspond to 0, or a non-zero integer of type `(signed-byte 32)`.
Values

\textit{int32}  
\textbf{An int32 object.}

Description

The function \textit{int32/} is the divide operator for \textit{int32} objects. See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also \textit{int32}

\textbf{int32/=}

\textit{Function}

Summary

The \textit{/=} comparison for \textit{int32} objects.

Package \textit{system}

Signature

\texttt{int32/= x y => result}

Arguments

\textit{x}  
\textbf{An int32 object or an integer of type (signed-byte 32).}

\textit{y}  
\textbf{An int32 object or an integer of type (signed-byte 32).}

Values

\textit{result}  
\textbf{A boolean.}

Description

The function \textit{int32/=} is the not equal comparison for \textit{int32} objects. See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also \textit{int32}

\textbf{int32<}

\textit{Function}

Summary

The \textit{<} comparison for \textit{int32} objects.
Package  

**system**

Signature  

```plaintext
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 less than comparison for `int32` objects.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also  

- `int32`

---

**int32<<**  

*Function*

Summary  

A shift left operator for `int32` objects.

Package  

**system**

Signature  

```plaintext
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.
See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also int32

**int32<=**

*Function*

**Summary**
The \( \leq \) comparison for int32 objects.

**Package**
*system*

**Signature**
\[
\text{int32} \leq \text{x y} \Rightarrow \text{result}
\]

**Arguments**
- \( x \):
  An int32 object or an integer of type \((\text{signed-byte 32})\).
- \( y \):
  An int32 object or an integer of type \((\text{signed-byte 32})\).

**Values**
- \( \text{result} \):
  A boolean.

**Description**
The function \( \text{int32} \leq \) is the less than or equal comparison for int32 objects.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also int32

**int32=**

*Function*

**Summary**
The \( = \) comparison for int32 objects.

**Package**
*system*

**Signature**
\[
\text{int32=} \text{x y} \Rightarrow \text{result}
\]
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 equal comparison for `int32` objects.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also `int32`

---

`int32>`  

Function

Summary

The `>` comparison for `int32` objects.

Package

`system`

Signature

`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 greater than comparison for `int32` objects.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.
See also int32

**int32>=**

*Function*

**Summary** The >= comparison for int32 objects.

**Package** system

**Signature** 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 greater than or equal comparison for int32 objects.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also int32

**int32>>**

*Function*

**Summary** A shift right operator for int32 objects.

**Package** system

**Signature** int32>> x y => result

**Arguments**

- **x** An int32 object or an integer of type (signed-byte 32).
**The SYSTEM Package**

An `int32` object or an integer of type 
(signed-byte 32).

Values

result  An int32 object.

Description

The function `int32>>` is a shift right operator for `int32` objects.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also  `int32`

---

**Function**

**int32-aref**

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 95 for more information about the INT32 API.
### int32-logand

**Function**

**Summary**
The `logand` operator for int32 objects.

**Package**
`system`

**Signature**
`int32-logand 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-logand` is the bitwise logical ‘and’ operator for int32 objects.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

**See also**
- int32

### int32-logandc1

**Function**

**Summary**
The `logandc1` operator for int32 objects.

**Package**
`system`

**Signature**
`int32-logandc1 x y => int32`
Arguments

\(x\)  An \texttt{int32} object or an integer of type \texttt{(signed-byte 32)}.

\(y\)  An \texttt{int32} object or an integer of type \texttt{(signed-byte 32)}.

Values

\texttt{int32}  An \texttt{int32} object.

Description

The function \texttt{int32-logandc1} is the bitwise logical operator for \texttt{int32} objects which ‘ands’ the complement of \(x\) with \(y\).See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also \texttt{int32}

\textbf{int32-logandc2}  \textit{Function}

Summary

The \texttt{logandc2} operator for \texttt{int32} objects.

Package \texttt{system}

Signature

\texttt{int32-logandc2} \(x\ y \Rightarrow \texttt{int32}\)

Arguments

\(x\)  An \texttt{int32} object or an integer of type \texttt{(signed-byte 32)}.

\(y\)  An \texttt{int32} object or an integer of type \texttt{(signed-byte 32)}.

Values

\texttt{int32}  An \texttt{int32} object.

Description

The function \texttt{int32-logandc2} is the bitwise logical operator for \texttt{int32} objects which ‘ands’ \(x\) with the complement of \(y\).See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.
int32-logbitp

Function

Summary The logbitp operator for int32 objects.

Package system

Signature int32-logbitp index x => result

Arguments

index An int32 object or an integer of type (signed-byte 32).

x An int32 object or an integer of type (signed-byte 32).

Values result An boolean.

Description The function int32-logbitp is the test for int32 objects which returns t if if the bit at index index in x is 1, and nil if it is 0.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also int32

int32-logeqv

Function

Summary The logeqv operator for int32 objects.

Package system

Signature int32-logeqv x y => int32
Arguments

\( x \)
An \texttt{int32} object or an integer of type (\texttt{signed-byte 32}).

\( y \)
An \texttt{int32} object or an integer of type (\texttt{signed-byte 32}).

Values

\texttt{int32}
An \texttt{int32} object.

Description

The function \texttt{int32-logeqv} is the bitwise logical operator for \texttt{int32} objects which returns the complement of the ‘exclusive or’ of \( x \) and \( y \).

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also

\texttt{int32}

\textbf{int32-logior}

Function

Summary

The \texttt{logior} operator for \texttt{int32} objects.

Package

\texttt{system}

Signature

\texttt{int32-logior} \( x \) \( y \) \( \Rightarrow \) \texttt{int32}

Arguments

\( x \)
An \texttt{int32} object or an integer of type (\texttt{signed-byte 32}).

\( y \)
An \texttt{int32} object or an integer of type (\texttt{signed-byte 32}).

Values

\texttt{int32}
An \texttt{int32} object.

Description

The function \texttt{int32-logior} is the bitwise logical ‘inclusive or’ operator for \texttt{int32} objects.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.
See also int32

int32-lognand

Function

Summary The lognand operator for int32 objects.

Package system

Signature int32-lognand 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-lognand is the bitwise logical operator for int32 objects which returns the complement of the ‘and’ of x and y.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also int32

int32-lognor

Function

Summary The lognor operator for int32 objects.

Package system

Signature int32-lognor x y => int32
The function `int32-lognor` is the bitwise logical operator for `int32` objects which returns the complement of the ‘inclusive or’ of `x` and `y`.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also `int32`

---

**int32-lognot**

**Function**

**Summary** The `lognot` operator for an `int32` object.

**Package** `system`

**Signature** `int32-lognot x => int32`

**Arguments**

- `x` An `int32` object or an integer of type `(signed-byte 32)`.

**Values**

- `int32` An `int32` object.

**Description** The function `int32-lognot` is the bitwise logical operator for `int32` objects which returns the complement of its argument `x`.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also `int32`
**int32-logorc1**

*Summary*  
The `logorc1` operator for `int32` objects.

*Package*  
`system`

*Signature*  
`int32-logorc1 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-logorc1` is the bitwise logical operator for `int32` objects which ‘inclusive ors’ the complement of `x` with `y`.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

*See also*  
`int32`

---

**int32-logorc2**

*Summary*  
The `logorc2` operator for `int32` objects.

*Package*  
`system`

*Signature*  
`int32-logorc2 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).`
### int32-logorc2

#### Description
The function `int32-logorc2` is the bitwise logical operator for `int32` objects which 'inclusive ors' `x` with the complement of `y`.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

#### See also
`int32`

---

### int32-logtest

#### Function

#### Summary
The `logtest` operator for `int32` objects.

#### Package
`system`

#### Signature
`int32-logtest 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 boolean.

#### Description
The function `int32-logtest` is the bitwise test for `int32` objects which returns `t` if any of the bits designated by 1 in `x` is 1 in `y`, and returns `nil` otherwise.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

#### See also
`int32`
**int32-logxor**

*Function*

**Summary**
The `logxor` operator for `int32` objects.

**Package**
`system`

**Signature**
`int32-logxor 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-logxor` is the bitwise logical 'exclusive or' operator for `int32` objects.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also

- `int32`

**int32-minusp**

*Function*

**Summary**
The `minusp` test for an `int32` object.

**Package**
`system`

**Signature**
`int32-minusp x => result`

**Arguments**

- `x`: An `int32` object or an integer of type `(signed-byte 32)`.

**Values**

- `result`: A boolean.
The function \texttt{int32-minusp} tests whether its argument \( x \) is \texttt{int32<} than the value of \texttt{+int32-0+}.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also \texttt{int32}

\begin{description}
\item[Description] The function \texttt{int32-plusp} tests whether its argument \( x \) is \texttt{int32>} than the value of \texttt{+int32-0+}.
\end{description}

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also \texttt{int32}

\begin{description}
\item[Description] The function \texttt{int32-plusp} tests whether its argument \( x \) is \texttt{int32>} than the value of \texttt{+int32-0+}.
\end{description}

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also \texttt{int32}

\begin{description}
\item[Description] The function \texttt{int32-plusp} tests whether its argument \( x \) is \texttt{int32>} than the value of \texttt{+int32-0+}.
\end{description}

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also \texttt{int32}

\begin{description}
\item[Description] The function \texttt{int32-plusp} tests whether its argument \( x \) is \texttt{int32>} than the value of \texttt{+int32-0+}.
\end{description}

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also \texttt{int32}

\begin{description}
\item[Description] The function \texttt{int32-plusp} tests whether its argument \( x \) is \texttt{int32>} than the value of \texttt{+int32-0+}.
\end{description}

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also \texttt{int32}

\begin{description}
\item[Description] The function \texttt{int32-plusp} tests whether its argument \( x \) is \texttt{int32>} than the value of \texttt{+int32-0+}.
\end{description}

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also \texttt{int32}

\begin{description}
\item[Description] The function \texttt{int32-plusp} tests whether its argument \( x \) is \texttt{int32>} than the value of \texttt{+int32-0+}.
\end{description}

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also \texttt{int32}
Arguments  $int32$  An $int32$ object or an integer of type \((\text{signed-byte\ }32)\).

Values  $integer$  An integer of type \((\text{signed-byte\ }32)\).

Description  The function $\text{int32-to-integer}$ returns an integer $integer$ of type \((\text{signed-byte\ }32)\) corresponding to the $int32$ object $int32$. The argument $int32$ can also be an integer of type \((\text{signed-byte\ }32)\), in which case it is simply returned.

An error is signalled if $int32$ is not of type $int32$ or \((\text{signed-byte\ }32)\).

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also  $int32$

---

**$int32$-zerop**

*Function*

Summary  The zerop test for an $int32$ object.

Package  \texttt{system}

Signature  $\text{int32-zerop}\ x \Rightarrow result$

Arguments  $x$  An $int32$ object or an integer of type \((\text{signed-byte\ }32)\).

Values  $result$  A boolean.

Description  The function $\text{int32-zerop}$ tests whether its argument $x$ is $int32\,$ to the value of $+\text{int32}\,0\,$.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also  $int32$
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.

Description  
The function integer-to-int32 constructs an int32 object from an integer. An error is signalled if integer is not of type (signed-byte 32).

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.

See also  
int32

*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 `strings` to compare them.

For a description of the command line arguments processed by LispWorks, see “The Command Line” on page 302.

See also `lisp-image-name`

### load-data-file

**Function**

**Summary**

Loads a fasl file created by `dump-forms-to-file` or `with-output-to-fasl-file`.

**Package**

`system`

**Signature**

`load-data-file file &rest args => result`

**Arguments**

- `file` A pathname designator.
- `args` Arguments passed to `load`.

**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 the same semantics as `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.

`load-data-file` is intended to work with data files generated in a previous version of LispWorks. In particular you can load data files generated by LispWorks 4.3, LispWorks 4.4 and LispWorks 5.0 into LispWorks 5.1.
Fasl files generated by `dump-forms-to-file` or `with-output-to-fasl-file` must only be loaded using `load-data-file`.

The pathname specified by `file` must be recognized as a fasl file type, otherwise `load-data-file` will load it as a text file.

**Compatibility Note**

The default fasl file type in LispWorks 5.0 and later differs to LispWorks 4.x on Windows and Linux, as described in `compile-file`. Therefore you may need to do something like this to ensure your LispWorks 4.x data file is recognized as a fasl file when loading it in this version of LispWorks:

```lisp
(let ((sys::*binary-file-types* (cons "fsl" sys::*binary-file-types*)))
  (sys:load-data-file "C:/temp/data.fsl"))
```

**Compatibility Note**

The `fixnum` type in LispWorks 5.0 and later is larger than in LispWorks 4.x on Windows and Linux. A `bignum` dumped in a LispWorks 4.x data file will be loaded as a `fixnum` in LispWorks 5.0 and later if its value is within the `fixnum` range.

See also `dump-forms-to-file` `with-output-to-fasl-file`

---

### locale-file-encoding

**Function**

Provides an encoding corresponding to the current code page on Microsoft Windows, and the locale on Unix.

**Summary**

**Package**

`system`

**Signature**

`locale-file-encoding pathname ef-spec buffer length => new-ef-spec`

**Arguments**

- `pathname` Pathname identifying location of `buffer`.
- `ef-spec` An external format spec.
- `buffer` A buffer whose contents are examined.
length: Length (an integer) up to which buffer should be examined.

Values: new-ef-spec: Default external format spec created by merging ef-spec with the encoding that was found.

Description: The function `locale-file-encoding` consults the ANSI code page on Microsoft Windows. If the code page identifier is in `win32:*latin-1-code-pages*`, `locale-file-encoding` merges ef-spec with :latin-1. This external format writes Latin-1 on output, giving an error for any non-Latin-1 characters that are written. If the code page identifier is not in `win32:*latin-1-code-pages*` then `locale-file-encoding` merges ef-spec with an encoding corresponding to the current code page that gives an error for characters that cannot be encoded.

`locale-file-encoding` merges ef-spec with :latin-1 on Unix.

See also: *file-encoding-detection-algorithm*  *latin-1-code-pages*  *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
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 175.

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 `char-code-limit`, or a Gesture Spec keyword, or nil.
- `modifiers` A non-negative integer less than 64, or nil.
- `can-shift-both-case-p` A generalized boolean.

**Values**
- `gspec` A Gesture Spec object
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
: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-hyper-bit`, `gesture-spec-meta-bit`, `gesture-spec-shift-bit`, and `gesture-spec-super-bit`. If *modifiers* is *nil*, then *gspec* has a wild modifiers component.
The gesture \texttt{Shift+X} could potentially be represented by the unmodified uppercase character \texttt{X}, or lowercase \texttt{x} with the \texttt{Shift} modifier. In order to ensure a consistent representation the latter form is not supported by Gesture Specs by default. That is, a \texttt{both-case-p} character may not be combined with the single modifier \texttt{Shift} in the accelerator argument. This can be overridden by passing a true value for \texttt{can-shift-both-case-p}.

A \texttt{both-case-p} character is allowed with \texttt{Shift} if there are other modifiers. See the below for examples.

Wild Gesture Specs can be useful when specifying an input model for a \texttt{capi:output-pane}.

\begin{example}
\begin{verbatim}
(sys:make-gesture-spec
  97
  (logior sys:gesture-spec-control-bit
   sys:gesture-spec-meta-bit))
\end{verbatim}
\end{example}

A \texttt{both-case-p} character may not be combined with the single modifier \texttt{Shift} in the accelerator argument, so code like this signals an error:

\begin{verbatim}
(sys:make-gesture-spec
  (char-code #\x)
  sys:gesture-spec-shift-bit)
\end{verbatim}

Instead you should use:

\begin{verbatim}
(sys:make-gesture-spec (char-code #\X) 0)
\end{verbatim}

A \texttt{both-case-p} character is allowed with \texttt{Shift} if there are other modifiers:

\begin{verbatim}
(sys:make-gesture-spec
  (char-code #\x)
  (logior sys:gesture-spec-shift-bit
   sys:gesture-spec-meta-bit))
\end{verbatim}

See also \texttt{gesture-spec-accelerator-bit} \texttt{gesture-spec-control-bit} \texttt{gesture-spec-data}
make-simple-int32-vector

Function

Summary
The constructor for simple-int32-vector objects.

Package
system

Signature
make-simple-int32-vector length &key initial-contents initial-element => vector

Arguments
length A non-negative fixnum.
initial-contents A sequence of integers of type (signed-byte 32), or nil.
initial-element An integer of type (signed-byte 32).

Values
vector A simple-int32-vector.

Description
The function make-simple-int32-vector is the constructor for simple-int32-vector objects.

The argument initial-contents, if supplied, should be a sequence of length length. It specifies the contents of vector.

The argument initial-element, if supplied, specifies the contents of vector.

An error is signalled if both initial-contents and initial-element are supplied.

See the section “Fast 32-bit arithmetic” on page 95 for more information about the INT32 API.
See also
int32
simple-int32-vector

**make-stderr-stream**

*Function*

Summary  Returns an output stream connected to stderr.

Package  system

Signature  make-stderr-stream => stream

Arguments  None.

Values  stream  An output stream.

Description  The function `make-stderr-stream` returns an output stream connected to stderr.

On Microsoft Windows, you should take care to not close this stream or make multiple stderr streams.

**make-typed-aref-vector**

*Function*

Summary  Makes a vector that can be accessed efficiently.

Package  system

Signature  make-typed-aref-vector byte-length => vector

Arguments  byte-length  A non-negative fixnum.

Values  vector  A vector.
The function `make-typed-aref-vector` returns a vector which is suitable for efficient access at compiler optimization level safety = 0.

Use `typed-aref` to access `vector` efficiently.

See also `typed-aref`

**Function**

### marking-gc

<table>
<thead>
<tr>
<th>Summary</th>
<th>Performs a Marking GC in 64-bit LispWorks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td><code>system</code></td>
</tr>
<tr>
<td>Signature</td>
<td><code>marking-gc gen-num &amp;key what-to-copy max-size max-size-to-copy fragmentation-threshold</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>gen-num</code> An integer in the inclusive range [0,7].</td>
</tr>
<tr>
<td></td>
<td><code>max-size-to-copy</code> A positive number or <code>nil</code>.</td>
</tr>
<tr>
<td></td>
<td><code>max-size</code> A synonym for <code>max-size-to-copy</code>.</td>
</tr>
<tr>
<td></td>
<td><code>fragmentation-threshold</code> A number in the inclusive range [0, 10].</td>
</tr>
<tr>
<td>Description</td>
<td>The function <code>marking-gc</code> garbage collects (GCs) the generation specified by <code>gen-num</code>, and all younger generations. It uses mark and sweep, rather than copy.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
The `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.
Note: this function 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

**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.

Note: `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`

**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.

Description The function merge-ef-specs returns an external format spec constructed by adding information not supplied in ef-spec1 from ef-spec2.

Each external format spec argument is either a symbol or a list.

If ef-spec1 and ef-spec2 have the same value for their name component (whether they are lists or symbols), return ef-spec1 combined with any parameters from ef-spec2 that are not specified in ef-spec1.

Otherwise, if ef-spec1 is :default or a list beginning with :default, return ef-spec2 with parameters modified to be a union of the parameters from ef-spec1 and ef-spec2, with those from ef-spec1 taking priority.

Otherwise, return ef-spec1 with any :eol-style parameter from ef-spec2 if ef-spec1 does not specify :eol-style.

---

**object-address**

Function

Summary Returns the address of the given object as an integer.

Package system

Signature object-address object => address
### Arguments

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>The object whose address should be returned.</td>
</tr>
</tbody>
</table>

### Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>The address of object. An integer.</td>
</tr>
</tbody>
</table>

### Description

Returns the address of the given object as an integer. Note that the address is likely to change during garbage collection so this integer should be used for debugging purposes only.

### Example

This shows that the address returned by `sys:object-address` is the same as the one printed by the `print-object` method for `generic-function`.

```lisp
CL-USER 1 > (let ((gf #'initialize-instance))
  (format t "address = ~X~%gf = ~S"
          (sys:object-address gf) gf))
address = 1cff778
gf = #<STANDARD-GENERIC-FUNCTION INITIALIZE-INSTANCE 1cfff778>
NIL

CL-USER 2 >
```

### See also

`pointer-from-address`

---

### open-pipe

#### Function

#### Summary

Runs a command in a subshell.

#### Package

`system`

#### Signature

`open-pipe command &key direction element-type interrupt-off shell-type => stream`

#### Arguments

<table>
<thead>
<tr>
<th>Name</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>direction</td>
<td>:input, :output or :io.</td>
</tr>
<tr>
<td>element-type</td>
<td>A type specifier.</td>
</tr>
</tbody>
</table>
interrupt-off A boolean. Not implemented on Microsoft Windows.

shell-type A shell type.

Values

stream A pipe stream.

Description

On Unix/Linux/Mac OS X 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.

If command is a string then it is passed to the shell as the command to run without any arguments. If command is a list, then its first element is the command to run directly and the other elements are passed as arguments on the command line (that is, element 0 has its name in argv[0] in C, and so on). If command is a simple-vector of strings, the element at index 0 is the command to run and the other elements are the complete set of arguments seen by the command (that is, element 1 becomes argv[0] in C, and so on). If command is nil, then the shell is run.

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. This argument is ignored on Microsoft Windows.

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 Lisp-Works 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/Linux/Mac OS X/FreeBSD 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".

stream supports mixed character and binary I/O in the same way as file streams constructed by open.

Examples

Example on Unix:

CL-USER 1 > (setf *ls* (sys:open-pipe "ls"))
Warning: Setting unbound variable *LS*
#<SYSTEM::PIPE-STREAM "ls">

CL-USER 2 > (loop while
    (print (read-line *ls* nil nil)))
"hello"
"othello"
NIL
NIL

CL-USER 3 > (close *ls*)
T

The following example shows you how to use bidirectional pipes.

CL-USER 1 > (with-open-stream
    (s (sys:open-pipe "/bin/csh"
        :direction :io))
    (write-line "whereis ls" s)
    (force-output s)
    (read-line s))
"ls: /sbin/ls /usr/bin/ls /usr/share/man/man1.Z/ls.1"
NIL

Example on Microsoft Windows
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

See also

 call-system
 call-system-showing-output

**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 Note**
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*`

---

**pid-exit-status**

*Function*

**Summary**
Returns the exit status of a process executed with `run-shell-command`.

**Package**
`system`

**Signature**
```
pid-exit-status pid &key wait name => exit-status
```

**Arguments**
- `pid`: A process ID.
- `wait`: A boolean, default value `t`.
- `name`: A Lisp object, default value `pid`.

**Values**
- `exit-status`: An integer, or `nil`.

**Description**
The function `pid-exit-status` returns the exit status of a process executed by `run-shell-command` with argument `save-exit-status` passed a non-nil value.

If `wait` is true then `pid-exit-status` waits until the process exits, using `name` in the wait message. If `wait` is `nil` and the process has not terminated, then `pid-exit-status` returns `nil` immediately.
Note: *pid-exit-status* is implemented only for Unix/Linux/Mac OS X.

See also run-shell-command

**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
Function

print-pretty-gesture-spec

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 objec 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.
**Variable**

*print-symbols-using-bars*

**Summary**

Controls how escaping is done when symbols are printed.

**Package**

system

**Initial Value**

nil

**Description**

The variable *print-symbols-using-bars* controls how escaping is done when symbols are printed.

When the value is true, printing symbols that must be escaped (for example, those with names containing the colon character :) is done using the bar character | instead of the backslash character \ in cases when the readtable case and *print-case* are both :upcase or both :downcase.

**Example**

```lisp
CL-USER 1 > readable-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**  

**Summary**  
Gets or sets a registry path for use with your software.

**Package**  
`system`

**Signature**  
`product-registry-path product => path-string`

**Signature**  
`(setf product-registry-path) path product => path`

**Arguments**  
`product`  
A Lisp object.

**Values**  
`path`  
The path as a string or a list of strings.

`path-string`  
The path as a string.

**Description**  
The function `product-registry-path` returns the registry subpath defined for the product denoted by `product`, as a string.

The function `(setf product-registry-path)` sets the registry subpath for the product denoted by `product`.

If `path` is a string it can contain backslash \ or forward slash / as directory separators - these are translated internally to the separator appropriate for the system. Note that any backslash will need escaping (with another backslash) if you input the string value via the Lisp reader.

If `path` is a list of strings, then it is interpreted like the directory component of a pathname.

This registry subpath is used when reading and storing user preferences with `user-preference`.

Note that while `product` can be any Lisp object, values of `product` are compared by `eq`, so you should use keywords.

**Note:** to store CAPI window geometries under the registry path for your product, see the entry for `capi:top-level-`
interface-geometry-key in the *LispWorks CAPI 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)`.

**Note:** In 64-bit LispWorks you can also use `count-gen-num-allocation` and `gen-num-segments-fragmentation-state`. 
run-shell-command

Function

Package

system

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

Arguments

command A string, a list of strings, a simple-vector of strings, or nil.

input nil, :stream or a file designator. Default value nil.

output nil, :stream or a file designator. Default value nil.

error-output nil, :stream, :output or a file designator. Default value nil.

separate-streams A boolean. True value not currently supported.

wait A boolean, default value t.

if-input-does-not-exist :error, :create or nil. Default value :error.


if-error-output-exists
The function run-shell-command allows 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 is 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` is `nil` and neither of `input` or `output` have the value `:stream` then the first return value, `stream`, is `nil`.

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 40.1 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><strong>input is :stream</strong></td>
<td>An I/O stream connected to stdin and stdout</td>
<td>An input stream connected to stderr</td>
</tr>
<tr>
<td><strong>output is :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>error-output is :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>input is not :stream</strong></td>
<td>An input stream connected to stdout</td>
<td>An input stream connected to stderr</td>
</tr>
<tr>
<td><strong>output is :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>error-output is :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>input is :stream</strong></td>
<td>An output stream connected to stdin</td>
<td>An input stream connected to stderr</td>
</tr>
<tr>
<td><strong>output is not :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>error-output is :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>input is not :stream</strong></td>
<td>nil</td>
<td>An input stream connected to stderr</td>
</tr>
<tr>
<td><strong>output is not :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>error-output is :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>input is :stream</strong></td>
<td>An I/O stream connected to stdin, stdout and stderr</td>
<td>nil</td>
</tr>
<tr>
<td><strong>output is :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>error-output is :output</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>input is not :stream</strong></td>
<td>An input stream connected to stdout and stderr</td>
<td>nil</td>
</tr>
<tr>
<td><strong>output is :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>error-output is :output</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>input is :stream</strong></td>
<td>An output stream connected to stdin</td>
<td>nil</td>
</tr>
<tr>
<td><strong>output is not :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>error-output is :output</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>input is not :stream</strong></td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td><strong>output is not :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>error-output is :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>input is :stream</strong></td>
<td>An I/O stream connected to stdin and stdout</td>
<td>nil</td>
</tr>
<tr>
<td><strong>output is :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>error-output is not :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>input is :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>output is not :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>error-output is not :stream</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>input is not :stream</strong></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 `pid-exit-status`.

**Note:** `run-shell-command` is implemented only for Unix/Linux/Mac OS X.
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

safe-locale-file-encoding

Function

Summary
Provides a safe encoding which corresponds to the current code page on Microsoft Windows, and the locale on Unix.

Package
system

Signature
```
safe-locale-file-encoding pathname ef-spec buffer length => new-ef-spec
```

Description
The function `safe-locale-file-encoding` is similar to `locale-file-encoding` except that it always returns a safe external format. That is, the external format does not signal error on writing characters not in the encoding.

On Microsoft Windows, `safe-locale-file-encoding` consults the ANSI code page. If the code page identifier `id` is in `win32:*latin-1-code-pages*`, it merges `ef-spec` with `:latin-1-safe`. This external format writes Latin-1 on output, using 63 (ASCII '?') to replace any non-Latin-1 characters that are written. If the code page identifier `id` is not in `win32:*latin-1-code-pages*` then `safe-locale-file-encoding` merges `ef-spec` with an encoding corresponding to the current code page.
that uses the code page's replacement code for characters that cannot be encoded.

`safe-locale-file-encoding` merges `ef-spec` with `:latin-1-safe` on Unix.

See also

*file-encoding-detection-algorithm*
*latin-1-code-pages*
locale-file-encoding

---

**set-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 block-
The SYSTEM Package

The automatic garbage collector (GC) in LispWorks automatically generates a blocking generation if the automatic GC is called by a method that is not a GC trap method. The generation number that was GCed can be obtained using the function `blocking-gen-num`. 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 GCed. It is called whenever an automatic GC occurred and `blocking-gen-num-func` was not called, either because the blocking generation was not GCed, 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.

**Note:** this function 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**

### Function

**set-blocking-gen-num**

**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`. 

See also

**set-blocking-gen-num**


**gc-threshold**  An integer greater than 12800, or a real in the inclusive range [0 100], or nil.

**Values**

- **old-blocking-gen-num**  An integer between 0 and 7, inclusive.
- **do-gc**  One of t, nil and :mark, or a real number between 0 and 10, inclusive.
- **max-size-to-copy**  A positive real number.
- **old-gc-threshold**  A number.

**Description**

The function `set-blocking-gen-num` sets `gen-num` as the generation that blocks. That is, no object is automatically promoted out of generation `gen-num` to a higher generation.

If `do-gc` is non-nil, then generation `gen-num` is automatically collected when needed, as defined by `gc-threshold` (see `set-gen-num-gc-threshold`).

The actual value of `do-gc` specifies how to GC the blocking generation when required. The possible values of `do-gc` are interpreted as follows:

- `t`  Use Copying GC.
- `:mark`  Use Marking GC.

A number in the inclusive range [0, 10]

Use Marking GC with copying of fragmented segments. The value specifies the `fragmentation-threshold` (the same as the argument to `marking-gc`). This is the ratio between the amount of free space that cannot be easily used and the amount of allocated space inside a segment. Only segments with fragmentation higher than the threshold are copied.

The default value of `do-gc` is `t`. 
max-size-to-copy is meaningful only if do-gc is a number. It specifies the maximum size in Gigabytes to try to copy. If the fragmented segments contain more data than this value, only some of them are copied in each GC.

If gc-threshold is non-nil, it is used to set the threshold for automatic GC using set-gen-num-gc-threshold.

The initial setup is as if this call has been made:

```
(sys:set-blocking-gen-num 3)
```

That is, the system will GC automatically according to the default gc-threshold using Copying GC.

Setting the blocking generation gen-num to a lower number is useful into two situations:

1. When you have an operation that allocates a significant amount of data, and almost of it goes when the operation finishes, it is useful to reduce the blocking gen-num during the operation. The macro block-promotion is a convenient way of doing that.

2. If you have a good idea of how your application behaves, it may be useful to block at a lower generation (2 or 1), and then periodically call 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.

Note: this function 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

set-default-segment-size

Function

Summary
Sets the default initial size of a segment in 64-bit LispWorks.

Package
system

Signature
set-default-segment-size gen-num allocation-type size-in-mb => segment-size

Arguments
gen-num An integer between 0 and 3, inclusive.


size-in-mb A number, or nil.

Values
segment-size A number.

Description
The function set-default-segment-size sets the default initial size of a segment for a specific generation and allocation type.
The default initial size is also used as the default size for enlargement of the segment.

`allocation-type` can be any of the allocation types. However, if `allocation-type` is `:other-big` or `:non-pointer-big`, this function has no effect.

If `size-in-mb` is a number, it specifies the size in megabytes. If `size-in-mb` is `nil` then `set-default-segment-size` returns the default initial segment size without altering it.

The returned value, `segment-size`, is the previous default initial segment size.

During automatic garbage collections (GCs) the system collects an ephemeral generation when any of its segments for the main allocation types is full. Thus the size of the segments defines the frequency of GCs in these generations.

**Note:** this function 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`

---

**set-delay-promotion**

**Function**

**Summary**

Delays promotion for a specified generation in 64-bit LispWorks.

**Package**

`system`

**Signature**

`set-delay-promotion gen-num on => on`

**Arguments**

- `gen-num` An integer between 0 and 7, inclusive.
- `on` A generalized boolean.
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.

**Note:** This function 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`
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 signalled on failure.

See also `open`

---

### `set-gen-num-gc-threshold` Function

**Summary**
Sets the additional allocation threshold that triggers a GC in the blocking generation in 64-bit LispWorks.

**Package**
`system`

**Signature**

```
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, \textit{threshold} can be an integer greater than 12800, which is interpreted as the absolute value. Alternatively, \textit{threshold} can be a real number in the inclusive range \([0 \, 100]\), which is multiplied by the allocation since the previous GC to get the actual threshold to set.

The default threshold for all generations is 1. That is, for all generations \textit{gen-num}, when generation \textit{gen-num} is the blocking generation and allocation in it has doubled since the previous GC, generation \textit{gen-num} is collected automatically.

\texttt{set-gen-num-gc-threshold} can be called when the generation \textit{gen-num} is not the blocking generation, and will set the value for that \textit{gen-num}. Such a call will not take effect until the generation \textit{gen-num} becomes the blocking generation, as set by a call to \texttt{set-blocking-gen-num} (with \texttt{:do-gc non-nil}).

Increasing the threshold reduces the number of GC calls, but may increase the virtual memory usage.

\texttt{set-gen-num-gc-threshold} returns the old threshold for the generation \textit{gen-num}. It can be called with \texttt{threshold nil} to return the threshold value without changing it.

\textbf{Note:} this function is implemented only in 64-bit LispWorks. It is not relevant to the Memory Management API in 32-bit implementations.

\textbf{See also} \texttt{set-blocking-gen-num}

\textbf{set-maximum-memory} \hspace{1cm} \textit{Function}

\textbf{Summary} \hspace{0.5cm} Sets or removes a limit for the top of the Lisp heap in 32-bit LispWorks.

\textbf{Package} \hspace{0.5cm} \textit{system}

\textbf{Signature} \hspace{0.5cm} \texttt{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 set-maximum-memory or the physical memory limit) the system will become unstable. Therefore this situation should be avoided. The benefit of having the maximum memory set is that a useful error is signaled if the limit is reached.

An application which is likely to grow to the maximum memory should test the amount of available memory using memory-growth-margin or room-values at suitable times, and take action to reclaim memory. Do not rely on handling the error signaled when the maximum memory is reached, since the system is already unstable at this point.

Note: set-maximum-memory is implemented only in 32-bit LispWorks. It is not relevant to the Memory Management API in 64-bit implementations.

See also

check-fragmentation
mark-and-sweep
memory-growth-margin
room-values
set-maximum-segment-size

Function

Summary
Defines the maximum segment size for a generation and allocation type in 64-bit LispWorks.

Package
system

Signature
set-maximum-segment-size gen-num allocation-type size-in-mb

Arguments

| gen-num | An integer between 0 and 7, inclusive. |
| 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 allocation types. However, if allocation-type is :other-big or :non-pointer-big, this function has no effect.

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 \texttt{marking-gc} with \texttt{what-to-copy non-nil} or \texttt{set-blocking-gen-num} with \texttt{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 \texttt{max-size-to-copy} parameter is set to a low number to avoid using too much virtual memory.

The returned value, \textit{max-segment-size}, is the previous maximum segment size.

If \texttt{size-in-mb} is a number, it specifies the size in megabytes. If \texttt{size-in-mb} is \texttt{nil} then \texttt{set-maximum-segment-size} returns the maximum segment size without altering it.

\textbf{Note:} this function is implemented only in 64-bit LispWorks. It is not relevant to the Memory Management API in 32-bit implementations.

\begin{verbatim}
See also  \texttt{marking-gc}
           \texttt{set-blocking-gen-num}
           \texttt{set-default-segment-size}
\end{verbatim}

---

\begin{verbatim}
set-memory-check  \hspace{1cm} Function

Summary  Sets a memory check in 64-bit LispWorks.

Package  system

Signature  \texttt{set-memory-check size function}

Arguments  \texttt{size} An integer.
            \texttt{function} A function designator.

Description  The function \texttt{set-memory-check} sets a memory check.
\end{verbatim}
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.

Note: this function 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

set-memory-exhausted-callback

Summary 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.
- `size`: The size in bytes which it was trying to allocate.
- `type-name`: A string naming the allocation type it was trying to allocate.
- `static`: A boolean, true if it was trying to allocate a static object, and false otherwise.

`function` can also have the special value `:reset`, which resets the callback list to `nil`.

`function` can also be `nil`, which means do nothing but simply return the current list of callbacks.

`where` defines the position in the list that the callback `function` is placed. Its allowed values are:

- `:first`: `function` is placed first in the callbacks list.
- `:last`: `function` is placed last in the callbacks list.
- `nil`: `function` is removed from the callbacks list.

`set-memory-exhausted-callback` always first removes `function` from the callbacks list, and then adds it according to
where. The default value of where is :first. Functions in the list are compared with equalp.

`set-memory-exhausted-callback` returns the callback list.

When a callback is called, Lisp already failed to map memory. This means that you must not rely on the callback to do real work. It should therefore attempt only a minimal amount of work such as clean-ups and generating debug information. It should not try to do real work.

After all the callbacks are called, the system signals an error of type `storage-exhausted`. The condition can be accessed using the accessors described for `storage-exhausted`.

**Note:** this function is implemented only in 64-bit LispWorks. It is not relevant to the Memory Management API in 32-bit implementations.

See also `set-memory-check` `storage-exhausted`

---

### set-signal-handler

**Function**

**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*.

**Example**

```
(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

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.

Note: this function is implemented only in 64-bit LispWorks. It is not relevant to the Memory Management API in 32-bit implementations.
See also  extended-time

**setup-atomic-funcall**  

**Function**

**Summary**  
Sets up mutually atomic funcalls in SMP LispWorks.

**Package**  
**system**

**Signature**  
`setup-atomic-funcall &rest function-and-arguments`

**Arguments**  
`function-and-arguments` A list.

**Description**  
The function `setup-atomic-funcall` sets up a funcall which will be executed atomically with respect to any other calls which were also set up by `setup-atomic-funcall`.

The call causes the execution of the form

```
(apply (car function-and-arguments)
       (cdr function-and-arguments))
```

some time after the entry to `setup-atomic-funcall`. The call may happen before `setup-atomic-funcall` returns, and it is expected that normally this is what will happen. However, it may be delayed for an indefinite period, but normally this period is short (milliseconds). The execution occurs atomically with respect to other calls that were set up by `setup-atomic-funcall`.

The call should be short, because otherwise it will delay all the other calls. If an error occurs during the call, the atomicity is no longer guaranteed.

`setup-atomic-funcall` is useful when a process needs to atomically tell another process to do something, but does not need to wait for it to finish.

`setup-atomic-funcall` causes less congestion than using a lock, and so is more efficient for locks that may cause conges-
tion. compare-and-swap and atomic-exchange operations will be faster.

See also
atomic-exchange
compare-and-swap

*sg-default-size*  

Variable

Summary  Default initial size of a stack group.

Package system

Initial Value
In LispWorks (64-bit) for Solaris:
20000
In all other implementations:
16000

Description  The value of the variable *sg-default-size* is the initial size of a stack group, in 32 bit words (in 32-bit implementations) or in 64 bit words (in 64-bit implementations).

*sg-default-size* can be bound around a call to a process creation function. Note that setting the global value of this variable affects the size of all system processes too, so this is not recommended.

Example
To create a process with a stack of 32000 words:

(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-augmented-string**

**Type**

Summary  The simple augmented string type.

Package  system

Signature  simple-augmented-string length

Arguments  length  The length of the string (or *, meaning any).

Description  This is the simple version of augmented-string, that is, the string itself is simple. Equivalent to:

(simple-vector character length)

See also  augmented-string

**simple-augmented-string-p**

**Function**

Summary  Tests if an object is a simple augmented string.

Package  system

Signature  simple-augmented-string-p object => bool

Arguments  object  The object to be tested.

Values  bool  t if object is a simple augmented string; nil otherwise.

Description  This is the predicate for simple augmented strings.

See also  simple-augmented-string
**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 95 for more information.

**See also**  
`int32`  
`int32-aref`  
`make-simple-int32-vector`

---

***stack-overflow-behaviour***  

**Variable**  

**Summary**  
Controls behavior when stack overflow occurs.

**Package**  
`system`

**Initial Value**  
`:error`

**Description**  
The variable `*stack-overflow-behaviour*` controls behavior when stack overflow occurs.

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.
Compatibility Note
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 instantiations 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  \textit{Function}

Summary  Applies a function to all the live objects in a generation in 64-bit LispWorks.

Package  system

Signature  sweep-gen-num-objects gen-num function

Arguments  
  gen-num An integer in the inclusive range $[0,7]$.
  function A designator for a function of one argument, the object.
Values

sweep-gen-num-objects returns nil.

Description

The function sweep-gen-num-objects applies function to all
the live objects in the generation gen-num.

function should take one argument, the object. It can allocate,
but if it allocates heavily the sweeping becomes unreliable.
Small amounts of allocation will normally happen only in
generation 0, and so will not affect sweeping of other genera-
tions.

Note: sweep-gen-num-objects is not implemented in 32-bit
LispWorks, where you can use sweep-all-objects instead.

See also

sweep-all-objects

typed-aref

Function

Summary

Accesses a typed aref vector efficiently.

Package

system

Signature

typed-aref type vector byte-index => value

(setf typed-aref) value type vector byte-index => value

Arguments

type A type specifier.

vector A vector created by make-typed-aref-vector.

byte-index A non-negative fixnum.

Values

value An object of type type.

Description

The function typed-aref allows efficient access to a typed
aref vector.

type must evaluate to one of: double-float, float, single-
float, sys:int32, (unsigned-byte 32), (signed-byte 32),
vector must be an object returned by make-typed-aref-vector.

byte-index specifies the index in 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.

Note: Efficient access to foreign arrays is also available. See fli:foreign-typed-aref in the LispWorks Foreign Language Interface User Guide and Reference Manual

Example

(defun double-float-typed-aref-incf (x y z)
  (declare (optimize (float 0) (safety 0)))
  (incf (sys:typed-aref 'double-float x y)
       (the double-float z))
  x)

See also make-typed-aref-vector

wait-for-input-streams Function

Summary
Waits for input on a list of socket streams, returning those that are ready.

Package system

Signature
wait-for-input-streams streams &key wait-function wait-reason timeout => result
### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>streams</td>
<td>A list, each member of which is a socket-stream.</td>
</tr>
<tr>
<td>wait-function</td>
<td>A function of no arguments.</td>
</tr>
<tr>
<td>wait-reason</td>
<td>A string.</td>
</tr>
<tr>
<td>timeout</td>
<td>A real number or nil.</td>
</tr>
</tbody>
</table>

### Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>A list of socket-streams or nil.</td>
</tr>
</tbody>
</table>

### Description

The function `wait-for-input-streams` waits for any of the streams in the argument `streams` to be ready for input. "Ready for input" typically means that some input is available from the stream, but can also mean that the peer closed the connection or there is an attempt to connect to the socket. Note that this function first checks the buffer for buffered streams.

When any of the streams is ready for input, `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.
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

Signature
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.
### The SYSTEM Package

**Package**  
`system`

**Signature**  
`with-modification-change modification-place &body body`

**Arguments**  
- `modification-place`: A place as defined in Common Lisp which can receive a fixnum.
- `body`: Lisp code

**Description**  
The macro `with-modification-change`, together with the macro `with-modification-check-macro`, provides a way for a body of code to execute and check whether there was any "modification" during this execution, where modification is execution of some other piece of code.

See “Aids for implementing modification checks” on page 176 for the full description and an example.

**Notes**  
`modification-place` does not need to be one of the places defined for low level atomic operations.

**See also**  
`with-modification-check-macro`

---

### with-modification-check-macro

**Macro**

**Summary**  
Provides a way to check whether there was any "modification" during execution of a body of code.

**Package**  
`system`

**Signature**  
`with-modification-check-macro macro-name modification-place &body body`

**Arguments**  
- `modification-place`: A place as defined in Common Lisp which can receive a fixnum.

**Description**  
The macro `with-modification-check-macro`, together with the macro `with-modification-change`, provides a way for a
body of code to execute and check whether there was any "modification" during this execution, where modification is execution of some other piece of code.

`with-modification-check-macro` defines a lexical macro (by macrolet) with the name `macro-name` which takes no arguments, and is used to check if 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 176 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 \texttt{WIN32} package. The \texttt{WIN32} package also includes functions for accessing the Microsoft Windows registry API, the DDE client interface, and the DDE server interface. These are documented in separate chapters in this manual.

\textbf{Note:} This chapter applies only to LispWorks for Windows, and not the UNIX, Linux, x86/x64 Solaris, FreeBSD or Mac OS X platforms.

\textbf{Note:} the \texttt{WIN32} package is not a supported implementation of the Win32 API. Define your own interfaces to Windows functions as you need - see the \textit{LispWorks Foreign Language Interface User Guide and Reference Manual} for details.

\textbf{dismiss-splash-screen} \textit{Function}

\begin{description}
\item[Summary] Makes a startup screen disappear.
\item[Package] \texttt{win32}
\item[Signature] \texttt{dismiss-splash-screen \&optional forcep}
\item[Arguments] \texttt{forcep} A generalized boolean.
\end{description}
**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*.

---

**Variable**

*latin-1-code-pages*  

**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.
This chapter applies only to LispWorks for Windows

Note: 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.
This chapter applies only to LispWorks for Windows

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

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 Windows XP, LispWorks is "themed", that is it uses the current theme of the desktop. You can switch this off by calling

(win32:set-application-themed nil)

On non-XP systems, or when the application does not have Common Controls 6, this call has no effect.

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, set-application-themed can be called multiple times in the same run.
short-namestring
Function

Summary
Returns the short form of a namestring.

Package
win32

Signature
short-namestring pathname => result

Arguments
pathname A pathname designator.

Values
result A string or nil.

Description
The function short-namestring first obtains the full namestring as if by cl:namestring, and then converts this namestring to the short form (in the Microsoft Windows meaning of “Short” paths).

If the translation succeeds then result is a string in the short form.

The translation may fail, in which case nil is returned.

See also
long-namestring

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
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 on Windows XP/Vista/7. Take care to interface to ANSI functions named like `FooBarA`, with the A suffix.

See also tstr

tstr
lpcstr
lpstr

Summary

Types which automatically switch between ANSI and Unicode strings.

Package

win32

Signature

tstr &key length
lpcstr &key max-length
lpstr &key max-length

Description

tstr is an ANSI/Unicode string.

lpcstr is a reference-pass pointer to ANSI/Unicode string.

lpstr 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`. 
This chapter applies only to LispWorks for Windows

Example

This calls `GetDriveTypeA` on Windows ME, and `GetDriveTypeW` on Windows XP/Vista/7.

The argument is passed as ANSI or Unicode respectively:

```lisp
(fli:define-foreign-function (%get-drive-type
"GetDriveType" :dbs)
  ((lpRootPathName W:LPCTSTR))
  :result-type (:unsigned :int))

(defconstant +drive-types+

(defun get-drive-information (drive)
  (the drive-type (svref +drive-types+ (%get-drive-type drive))))
```

### Types converting to Unicode strings.

<table>
<thead>
<tr>
<th>FLI type descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>wstr</strong></td>
</tr>
<tr>
<td><strong>lpcwstr</strong></td>
</tr>
<tr>
<td><strong>lpwstr</strong></td>
</tr>
</tbody>
</table>

**Summary**

- **wstr** is an Unicode string.
- **lpcwstr** is a reference-pass pointer to an Unicode string.
- **lpwstr** is a reference (in/out) pointer to an Unicode string.

These three types are Unicode only. You are unlikely to need these unless you know your application only needs to run on Windows XP/Vista/7, or if you are interfacing to some of the few 'W' functions that are available on Windows ME. In that
case you need to pass the correct function name, something like `FooBarW` with the `W` suffix, to `fli:define-foreign-function`.

See also `tstr`
This chapter describes the Microsoft Windows registry API, which is available in the WIN32 package.

The WIN32 package also includes functions for accessing miscellaneous Windows functionality, the DDE client interface, and the DDE server interface. These are documented in separate chapters in this manual.

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.

**Signature**

\[
\text{close-registry-key} \ handle \ &key \ errorp \Rightarrow successp, \ error-code
\]

**Arguments**

- `handle`: A handle to an open registry key.

**Values**

- `successp`: A boolean.
- `error-code`: An integer error code or `nil`. 
The function **close-registry-key** closes `handle`, which should be an open registry key handle.

The return value on success is `t`.

If an error occurs and `errorp` is true then an error is signalled. Otherwise, the return values are `nil` and the Windows `error-code`. The default value of `errorp` is `t`.

**See also**

- `create-registry-key`
- `open-registry-key`

---

**collect-registry-subkeys**

**Function**

**Summary**

Returns names of the subkeys of a registry key.

**Signature**

```lisp
collect-registry-subkeys subkey &key root max-name-size max-names errorp value-function => subsubkeys
```

**Arguments**

- `subkey` A string specifying the name of the key.
- `root` A keyword or handle.
- `max-name-size` An integer.
- `max-names` An integer.
- `errorp` A boolean.
- `value-function` A function designator or `nil`.

**Values**

- `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 signalled. 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 subsubkey-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 signalled. Otherwise, subsubkeys is returned as nil if subkey could not be opened. The default value of errorp is t.

See also collect-registry-values create-registry-key

**collect-registry-values**

*Function*

**Summary**

Returns the values of a registry key.

**Signature**

collect-registry-values subkey &key root max-name-size max-buffer-size expected-type errorp value-function => values-alist

**Arguments**

subkey A string specifying the name of the key.

root A keyword or handle.

max-name-size An integer.
max-buffer-size  An integer.

expected-type    A keyword or t.

errorp          A boolean.

value-function A function or symbol.

Values values-alist An alist.

Description The function \texttt{collect-registry-values} returns an alist of all of the values of \texttt{subkey} under the key \texttt{root}.

\texttt{subkey} and \texttt{root} are interpreted as described for \texttt{create-registry-key}. The default value of \texttt{root} is \texttt{:user}.

\texttt{max-name-size} specifies the maximum length of the returned name. If the name is longer than this, an error is signalled. The default value of \texttt{max-name-size} is 256.

\texttt{max-buffer-size} specifies the maximum length in bytes of the data. If the data is longer than this, an error is signalled. The default value of \texttt{max-buffer-size} is 1024.

If \texttt{value-function} is \texttt{nil}, the returned \texttt{values-alist} is an association list containing pairs \texttt{(name . data)} consisting of the names and data of the values of \texttt{subkey}. \texttt{expected-type} controls how certain types are converted to Lisp objects as described for \texttt{enum-registry-value}. The default value of \texttt{expected-type} is \texttt{t}.

If \texttt{value-function} is non-nil, it should be a function with signature

\texttt{value-function handle subsubkey-name-and-value => name-and-value, collectp}

\texttt{value-function} is funccalled for each subsubkey with the handle of \texttt{subkey} and a cons of the name and value of the subsubkey. If \texttt{collectp} is non-nil then \texttt{name-and-value} is collected into the alist \texttt{values-alist} to return from \texttt{collect-registry-values}. Otherwise \texttt{name-and-value} is ignored.
If an error occurs and errorp is true, then an error is signalled. 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**
Create a new registry key.

**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.
- `error-code`: An integer error code or nil.

**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 or :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 signalled. 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.

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.
This chapter applies only to LispWorks for Windows

\textbf{error-code} \hspace{1cm} \text{An integer error code or nil.}

\textbf{Description} \hspace{1cm} The function \texttt{delete-registry-key} deletes the registry key named \texttt{subkey} under the parent key \texttt{root}.

\texttt{subkey} and \texttt{root} are interpreted as described for \texttt{create-registry-key}. The default value of \texttt{root} is \texttt{:user}.

The value \texttt{t} is returned if the key is deleted successfully.

If an error occurs and \texttt{errorp} is true then an error is signalled. Otherwise, the return values are \texttt{nil} and the Windows \texttt{error-code}. The default value of \texttt{errorp} is \texttt{t}.

\textbf{See also} \hspace{1cm} \texttt{create-registry-key}

\textbf{enum-registry-value} \hspace{1cm} \textit{Function}

\textbf{Summary} \hspace{1cm} Enumerates the values of a registry key.

\textbf{Signature} \hspace{1cm} \texttt{enum-registry-value subkey index &key root max-name-size max-buffer-size expected-type errorp => name, data-type, data, error-code}

\textbf{Arguments} \hspace{1cm} \begin{itemize}
  \item \texttt{subkey} \hspace{1cm} A string specifying the name of the key.
  \item \texttt{index} \hspace{1cm} An integer.
  \item \texttt{root} \hspace{1cm} A keyword or handle.
  \item \texttt{max-name-size} \hspace{1cm} An integer.
  \item \texttt{max-buffer-size} \hspace{1cm} An integer.
  \item \texttt{expected-type} \hspace{1cm} A keyword or \texttt{t}.
  \item \texttt{errorp} \hspace{1cm} A boolean.
\end{itemize}

\textbf{Values} \hspace{1cm} \begin{itemize}
  \item \texttt{name} \hspace{1cm} A string.
  \item \texttt{data-type} \hspace{1cm} A keyword.
  \item \texttt{data} \hspace{1cm} A lisp object.
\end{itemize}
The function `enum-registry-value` allows the values of sub-key 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 signalled. 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 signalled. 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` controls how certain data types are converted to Lisp objects as follows:

<table>
<thead>
<tr>
<th>data-type</th>
<th>expected-type</th>
<th>Description of converted data</th>
</tr>
</thead>
<tbody>
<tr>
<td>:string</td>
<td>:lisp-object</td>
<td>String made with <code>read-from-string</code></td>
</tr>
<tr>
<td>:string</td>
<td>Not supplied</td>
<td>String, exactly as in the registry</td>
</tr>
<tr>
<td>:environment-string</td>
<td>:string</td>
<td>String, exactly as in the registry</td>
</tr>
<tr>
<td>:environment-string</td>
<td>Not supplied</td>
<td>String, environment variables expanded</td>
</tr>
<tr>
<td>:integer</td>
<td>Not supplied</td>
<td>Integer</td>
</tr>
</tbody>
</table>
This chapter applies only to LispWorks for Windows

The default value of `expected-type` is `t`.

If an error occurs and `errorp` is true, then an error is signalled. 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.

**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`.

---

### Table 42.1 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>: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 signalled. Otherwise, the return values are `nil`, `nil`, `nil` and the Windows `error-code`. The default value of `errorp` is `t`. 
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 signalled. Otherwise, the return values are nil and the Windows error-code. The default value of errorp is t.

See also create-registry-key

query-registry-key-info

Summary Returns information about an open registry key handle.

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.
This chapter applies only to LispWorks for Windows

: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.

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 signalled. 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.

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.

**Signature**

```lisp
registry-value subkey name &key root expected-type errorp =>
  data, successp, error-code

(setf registry-value) value subkey name &key root expected-type errorp => value
```

**Arguments**

- **subkey**
  - A string specifying the name of the key.
- **name**
  - A string specifying the name of the value.
- **root**
  - A keyword or handle.
- **expected-type**
  - A keyword or t.
- **errorp**
  - A boolean.

**Values**

- **data**
  - A Lisp object.
- **successp**
  - A boolean.
- **error-code**
  - An integer error code or nil.

**Description**

The function **registry-value** returns the value associated with *name* in *subkey* under the key *root*.
subkey and root are interpreted as described for `create-registry-key`. The default value of root is :user.

If the value exists, then the return values are the data and true. expected-type controls how certain types are converted to Lisp objects as described for `enum-registry-value`. The default value of expected-type is t.

If an error occurs and errorp is true then an error is signalled. Otherwise, the return values are nil, nil and the Windows error-code. The default value of errorp is t.

The function (setf registry-value) sets the value associated with name in subkey under the key root, creating the subkey if necessary. The default value of root is :user.

See also `set-registry-value`

---

**set-registry-value**

**Function**

**Summary**
Stores a value in the registry.

**Signature**

```
(set-registry-value data subkey name &key root expected-type errorp => error-code)
```

**Arguments**

- **data**
  A Lisp object.

- **subkey**
  A string specifying the name of the key.

- **name**
  A string specifying the name of the value.

- **root**
  A keyword or handle.

- **expected-type**
  A keyword or t.

- **errorp**
  A boolean.

**Values**

- **error-code**
  An integer error code or nil.

**Description**

The function `set-registry-value` sets the value associated with name in subkey under the key root.
subkey and root are interpreted as described for create-registry-key. The default value of root is :user.

The stored value is derived from data, converted according to expected-type as follows:

<table>
<thead>
<tr>
<th>Lisp data</th>
<th>expected-type</th>
<th>Registry type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A string</td>
<td>:string</td>
<td>REG_SZ exactly as in data</td>
</tr>
<tr>
<td>Lisp value</td>
<td>:lisp-object</td>
<td>REG_SZ made with prin1-to-string of data</td>
</tr>
<tr>
<td>An integer</td>
<td>:integer</td>
<td>REG_DWORD containing data</td>
</tr>
<tr>
<td>A foreign pointer</td>
<td>:binary</td>
<td>REG_BINARY containing bytes of one element at the pointer</td>
</tr>
<tr>
<td>An array</td>
<td>:binary</td>
<td>REG_BINARY containing bytes from the array</td>
</tr>
</tbody>
</table>

The default value of expected-type is t.

If an error occurs and errorp is true then an error is signalled. The default value of errorp is t.

See also create-registry-key registry-value

with-registry-key

Summary Runs code with an open registry key handle.
The Windows registry API

This chapter applies only to LispWorks for Windows

**Signature**

```
with-registry-key (handle subkey &key root access errorp)
&body body => values
```

**Arguments**

- **handle**: A variable name.
- **subkey**: A string specifying the name of the key.
- **root**: A keyword or handle.
- **access**: An integer or keyword.
- **errorp**: A boolean.

**Values**

- **values**: The values returned by `body`.

**Description**

The macro `with-registry-key` evaluates `body` with the variable `handle` bound to the registry key handle opened as if by calling

```
(open-registry-key subkey :root root
 :access access
 :errorp errorp)
```

`subkey` and `root` are interpreted as described for `create-registry-key`.

If `errorp` is `nil` and `subkey` cannot be opened then `body` is not evaluated.

**See also**

`create-registry-key`
The DDE client interface

This chapter describes the Dynamic Data Exchange (DDE) client interface which is available in the *WIN32* package. You should use this chapter in conjunction with Chapter 18, “Dynamic Data Exchange”.

The *WIN32* package also includes functions for accessing miscellaneous Microsoft Windows functionality, the registry API, and the DDE server interface. These are documented in separate chapters in this manual.

**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.

**Package**
*win32*

**Signature**

```lisp
dde-advice-start conversation item &key key function format datap type errorp => result
```

**Arguments**

- `conversation` A conversation object.
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.

Values  result  A boolean.

Description  The **dde-advise-start** function sets up an advise loop for the data item specified by *item* on the specified *conversation*.

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 *type* specifies how the response data should be converted to a Lisp object. For text formats, the default value indicates that a Lisp string should be created. The value ':string-list' may be specified to indicate that the return value should be taken as a tab-separated list of strings; in this case the Lisp return value is a list of strings. The default conversation class only supports text formats, unless *type* is specified as ':foreign', which can be used with any clipboard format.
format. It returns a clipboard-item structure, containing a foreign pointer to the data, the data length, and the format identifier.

If \( \text{datap} \) is \( \text{t} \) (the default value), a hot link is established, where the new data is supplied whenever it changes. If \( \text{datap} \) is \( \text{nil} \), a warm link is established, where the data is not passed, and must be explicitly requested using \text{dde-request}.

The argument \text{key} is used to identify this link. If specified as \( \text{nil} \) (the default value), it defaults to the conversation. Multiple links are permitted on a conversation with the same \text{item} and \text{format} values, as long as their \text{key} values differ.

If the link is established, the return value \text{result} is \( \text{t} \). If the link could not be established, the behavior depends on the value of \text{errorp}. If \text{errorp} is \( \text{t} \) (the default value), LispWorks signals an error. If it is \( \text{nil} \), the function returns \( \text{nil} \) to indicate failure.

If the link is established, the function \text{function} is called whenever the data changes. If \text{function} is \( \text{nil} \) (the default value), then the generic function \text{dde-client-advise-data} will be called.

The function specified by \text{function} should have a lambda list similar to the following:

\[
\text{key} \quad \text{item} \quad \text{data} \quad \& \text{key} \quad \text{conversation} \quad \& \text{allow-other-keys}
\]

The arguments \text{key} and \text{item} identify the link. The argument \text{data} contains the new data for hot links; for warm links it is \( \text{nil} \).

See also

\text{dde-advise-start*}
\text{dde-advise-stop}
\text{dde-client-advise-data}
**Function**

### dde-advice-start*

**Summary**
Sets up an advise loop for a specified data item for an automatically managed conversation.

**Package**
`win32`

**Signature**
```
dde-advice-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 `dde-advice-start*` function is similar to the `dde-advice-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 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-advise-data* will be called.

The function specified by *function* should have a lambda list similar to the following:

```
key item data &key conversation &allow-other-keys
```

The arguments *key* and *item* identify the link. The argument *data* contains the new data for hot links; for warm links it is *nil*.

See also  
dde-advise-start  
dde-advise-stop  
dde-advise-stop*  
dde-client-advise-data

### Function

dde-advise-stop

**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-advice-stop` removes a particular link from a 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.

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-advice-start`
- `dde-advice-start*`
- `dde-advice-stop*`
- `dde-client-advice-data`

### Function

#### dde-advice-stop*

**Summary**

Removes a link from an automatically managed conversation specified by a given item and key.
This chapter applies only to LispWorks for Windows

Package  
**win32**

Signature  
```
dde-advice-stop* service topic item &key key format errorp disconnectp => result
```

Arguments  
- *service* A string or symbol.
- *topic* A string or symbol.
- *item* A string or symbol.
- *key* An object.
- *format* A clipboard format specifier.
- *errorp* A boolean.
- *disconnectp* A boolean.

Values  
- *result* A boolean.

Description  
The function `dde-advice-stop*` is similar to the function `dde-advice-stop`, and removes a particular link from a conversation specified by the `service/topic` pair indicated by `item`, `format` and `key`. If `key` is the last key for the `item/format` pair, the advise loop for the pair is terminated.

If `disconnectp` is `t` (the default value), and the last advise loop for the conversation is terminated, the conversation is disconnected.

If this function succeeds, it returns `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-advice-start`
- `dde-advice-start*`
- `dde-advice-stop`
**dde-client-advise-data**

*Generic Function*

**Summary**
Called when data changes in an advise loop.

**Package**
`win32`

**Signature**
`dde-client-advise-data key item data &key &allow-other-keys`

**Arguments**
- `key`: An object.
- `item`: A string or symbol.
- `data`: A string.

**Values**
None.

**Description**
The generic function `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 `dde-advertise-start` or `dde-advertise-start*`, or on a client conversation class if the default `key` is used.

**See also**
- `dde-advertise-start`
- `dde-advertise-stop`

---

**dde-connect**

*Function*

**Summary**
Attempts to create a conversation with a specified DDE server.

**Package**
`win32`

**Signature**
`dde-connect service topic &key class errorp => object`

**Arguments**
- `service`: A symbol or string.
This chapter applies only to LispWorks for Windows

<table>
<thead>
<tr>
<th>Values</th>
<th>object</th>
<th>A conversation object.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>topic</strong></td>
<td>A symbol or string.</td>
<td></td>
</tr>
<tr>
<td><strong>class</strong></td>
<td>The class of the conversation object to create.</td>
<td></td>
</tr>
<tr>
<td><strong>errorp</strong></td>
<td>A boolean.</td>
<td></td>
</tr>
</tbody>
</table>

**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 (lightweight process) in which they were created.

**See also**

- **dde-disconnect**

**dde-disconnect**

**Summary**

Disconnects a conversation object.
### dde-disconnect

<table>
<thead>
<tr>
<th>Package</th>
<th>win32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>dde-disconnect conversation =&gt; result</td>
</tr>
<tr>
<td>Arguments</td>
<td>conversation</td>
</tr>
<tr>
<td>Values</td>
<td>result</td>
</tr>
<tr>
<td>Description</td>
<td>The function dde-disconnect disconnects the conversation object. The conversation may no longer be used. If the conversation disconnects successfully, t is returned.</td>
</tr>
<tr>
<td>See also</td>
<td>dde-connect</td>
</tr>
</tbody>
</table>

### dde-execute

<table>
<thead>
<tr>
<th>Package</th>
<th>win32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>dde-execute conversation command &amp;rest {args}* =&gt; result</td>
</tr>
<tr>
<td>Arguments</td>
<td>conversation</td>
</tr>
<tr>
<td></td>
<td>command</td>
</tr>
<tr>
<td></td>
<td>args</td>
</tr>
<tr>
<td>Values</td>
<td>result</td>
</tr>
<tr>
<td>Description</td>
<td>The function dde-execute provides an alternative syntax for dde-execute-command. Unlike dde-execute-command, dde-execute takes the arguments for command as a sequence of args following &amp;rest, and does not have an argument for specifying how to handle an error.</td>
</tr>
</tbody>
</table>
This chapter applies only to LispWorks for Windows

See also
dde-execute*
dde-execute-command*
dde-execute-string

dde-execute*

Function

Summary
An alternative syntax for dde-execute-command*.

Package
win32

Signature
dde-execute* service topic command &rest {args}* => result

Arguments
service A string or symbol.

topic A string symbol.

command A string or symbol.

args An argument.

Values
result A boolean.

Description
The function dde-execute* provides an alternative syntax for dde-execute-command*. Unlike dde-execute-command*, dde-execute* takes the arguments for 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

\[[\text{command}(\text{arg1, arg2, ...})]\]

On success, this function returns a result of t. On failure, the behavior depends on the value of the errorp argument. If errorp is t (the default value), LispWorks signals an error. If it is nil, the function returns nil to indicate failure.

See also dde-execute
dde-execute-string

dde-execute-command*

Function

Summary Sends a command string to a specified service on a given topic.

Package win32
This chapter applies only to LispWorks for Windows

**Signature**

```
dde-execute-command* service topic command arg-list &key errorp connect-error-p new-conversation-p => result
```

**Arguments**

- `service` A string or symbol.
- `topic` A string or symbol.
- `command` A string or symbol.
- `arg-list` A list of strings, integers, and floats.
- `errorp` A boolean.
- `connect-error-p` A boolean.
- `new-conversation-p` A boolean.

**Values**

- `result` A boolean.

**Description**

The function `dde-execute-command*` is similar to `dde-execute-command`, and sends a command string to the server specified by `service` on a topic given by `topic`. The command string consists of `command` and `arg-list`, which are combined using the appropriate argument-marshalling conventions. By default, the syntax is

```
[command(arg1, arg2, ...)]
```

If `server` names a client service registered with `define-dde-client`, the registered service name is used as the DDE service name. If `server` is any other symbol, the print name of the symbol is used as the DDE service name. If `server` is a string, that string is used as the DDE service name.

The `topic` argument specifies the DDE topic name to be used in the conversation. If it is a symbol, the symbol’s print name is used. If it is a string, the string is used.

If necessary, the function `dde-execute-command*` creates a conversation for the duration of the transaction, but if a suitable conversation already exists, the transaction is executed...
over that conversation. Hence, if several transactions will be made with the same service and topic, placing them inside a with-dde-conversation prevents a new conversation being established for each transaction.

If new-conversation-p is set to t a new conversation is always established for the transaction. This new conversation is always automatically disconnected when the transaction is completed.

If connect-error-p is t (the default value) and a conversation cannot be established, then LispWorks signals an error. If it is nil, dde-execute-command* returns nil if a conversation cannot be established. This allows the caller to distinguish between the cases when the server is not running, and when the server is running but the transaction fails.

Upon success, this function returns a result of t. On failure, the behavior depends on the value of the errorp argument. If errorp is t (the default value), LispWorks signals an error. If it is nil, the function returns nil to indicate failure.

See also

dde-execute

dde-execute-string

dde-execute-command

dde-execute-string

**Function**

**Summary**

Issues an execute transaction consisting of a specified string.

**Package**

win32

**Signature**

dde-execute-string conversation command &key errorp => result

**Arguments**

conversation A conversation object.

command A string or symbol.

errorp A boolean.
This chapter applies only to LispWorks for Windows

### Values

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>A boolean.</td>
</tr>
</tbody>
</table>

### Description

The 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 211. 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`

**Signature**

```
(dde-execute-string* service topic command &key errorp
  connect-error-p new-conversation-p => result)
```

**Arguments**

- `service` A symbol or string.
- `topic` A symbol or string.
- `command` A string or symbol.
- `errorp` A boolean.
- `connect-error-p` A boolean.
- `new-conversation-p` A boolean.
Values

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
</table>
| The function `dde-execute-string*` is similar to `dde-execute-string`, in that it issues an execute transaction consisting of the string `command`. However, the conversation across which `command` is issued is managed automatically. No processing of the string is performed.

If `server` names a client service registered with `define-dde-client`, the registered service name is used as the DDE service name. If `server` is any other symbol, the print name of the symbol is used as the DDE service name. If `server` is a string, that string is used as the DDE service name.

The `topic` argument specifies the DDE topic name to be used in the conversation. If it is a symbol, the symbol's print name is used. If it is a string, the string is used.

If necessary, the function `dde-execute-string*` will create a conversation for the duration of the transaction, but if a suitable conversation already exists, the transaction will be executed over that conversation. Hence, if several transactions will be made with the same `service` and `topic`, placing them inside a `with-dde-conversation` prevents a new conversation being established for each transaction.

If `new-conversation-p` is set to `t` a new conversation is always established for the transaction. This new conversation is always automatically disconnected when the transaction is completed.

If `connect-error-p` is `t` (the default value), then LispWorks signals an error if a conversation cannot be established. If it is `nil`, `dde-execute-string*` returns `nil` if a conversation cannot be established. This allows the caller to distinguish between the cases when the server is not running, and when the server is running but the transaction fails.

Upon success, the function returns `t`. On failure, the behavior depends on the value of the `errorp` argument. If `errorp` is `t` (the
default value), LispWorks signals an error. If it is nil, the function returns nil to indicate failure.

See also

- `dde-execute`
- `dde-execute-command`
- `dde-execute-string`

**dde-item**  
*Accessor*

**Summary**  
An accessor which can perform a request transaction or a poke transaction.

**Package**  
win32

**Signature**  
```
dde-item conversation item &key format type errorp => result
```

**Arguments**  
- `conversation`  
  A conversation object.
- `item`  
  A string or symbol.
- `format`  
  A clipboard format specifier.
- `type`  
  A keyword.
- `errorp`  
  A boolean.

**Values**  
- `result`  
  A boolean.

**Description**  
The accessor `dde-item` performs a request transaction when read. It performs a poke transaction when set. To illustrate, the following `dde-request` command

```
(dde-request conversation item :format format :type type :errorp errorp)
```

can also be issued using `dde-item` as follows:

```
(dde-item conversation item :FORMAT format :TYPE type :ERRORP errorp)
```
Similarly, the following `dde-poke` command

```
(dde-poke conversation item data :format format :type type :errorp errorp)
```

can be issued using `dde-item` as follows:

```
(setq (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`

**Signature**  
`dde-item* service topic item &key format type errorp connect-error-p new-conversation-p => result`

**Arguments**  
- `service`: A string or symbol.
- `topic`: A string or symbol.
- `item`: A string or symbol.
- `format`: A clipboard format specifier.
- `type`: A keyword.
- `errorp`: A boolean.
This chapter applies only to LispWorks for Windows

connect-error-p  A boolean.

new-conversation-p  A boolean.

Values  result  A boolean.

Description  The accessor dde-item* is similar to dde-item, and performs a request transaction when read. It performs a poke transaction when set.

To illustrate, the following dde-request* command

(dde-request* service topic item :format format :type type :errorp errorp connect-error-p new-conversation-p)

can also be issued using dde-item* as follows:

(dde-item* service topic item :FORMAT format :TYPE type :ERRORP errorp connect-error-p new-conversation-p)

Similarly, the following dde-poke* command

(dde-poke* conversation item data :format format :type type :errorp errorp connect-error-p new-conversation-p)

can be issued using dde-item* as follows:

(setf (dde-item* conversation item :format format :type type :errorp errorp connect-error-p new-conversation-p) data)

except that the format always returns data.

If necessary, the accessor dde-item* creates a conversation for the duration of the transaction, but if a suitable conversation already exists, the transaction is executed over that conversation. If you need to make several transactions with the same service and topic, placing them inside a with-dde-conversation prevents a new conversation being established for each transaction.

If new-conversation-p is set to t a new conversation is always established for the transaction. This new conversation is
always automatically disconnected when the transaction is completed.

If `connect-error-p` is t (the default value), then LispWorks signals an error if a conversation cannot be established. If it is `nil`, `dde-item*` returns `nil` if a conversation cannot be established. This allows the caller to distinguish between the cases when the server is not running, and when the server is running but the transaction fails.

On success, the function returns t. On failure, the behavior depends on the value of the `errorp` argument. If `errorp` is t (the default value), LispWorks signals an error. If it is `nil`, the function returns `nil` to indicate failure.

See also

- `dde-item`
- `dde-poke`
- `dde-request`

### `dde-poke`

**Function**

**Summary** Issues a poke transaction on a conversation, to set the value of a specified item.

**Package** \texttt{win32}

**Signature**

\begin{verbatim}
dde-poke conversation item data &key format type errorp => result
\end{verbatim}

**Arguments**

- `conversation` A conversation object.
- `item` A string or symbol.
- `data` A string.
- `format` A clipboard format specifier.
- `type` A keyword.
- `errorp` A boolean.
This chapter applies only to LispWorks for Windows

Values

| Values | result | A boolean. |

Description

The function **dde-poke** issues a poke transaction on *conversation* 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.

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.

For text transactions, the default value of *type* indicates that *data* is a Lisp string to be used. If *type* is :string-list, then *data* is taken to be a list of strings, and is sent as a tab-separated string.

Alternatively, *data* can be a clipboard-item structure, containing a foreign pointer to the data to send and the length of the data. In this case the *type* argument is ignored.

On success, this function returns `t`. On failure, the behavior depends on the value of the *errorp* argument. If *errorp* is `t` (the default value), LispWorks signals an error. If it is `nil`, the function returns `nil` to indicate failure.

See also

- **dde-item**
- **dde-request**
**dde-poke**

**Function**

**Summary**
Issues a poke transaction on an automatically managed conversation, to set the value of a specified item.

**Package**
win32

**Signature**
dde-poke* service topic item data &key format type errorp connect-error-p new-conversation-p => result

**Arguments**
- service: A symbol or string.
- topic: A symbol or string.
- item: A string or symbol.
- data: A string.
- format: A clipboard format specifier.
- type: A keyword.
- errorp: A boolean.
- connect-error-p: A boolean.
- new-conversation-p: A boolean.

**Values**
- result: A boolean.

**Description**
The function **dde-poke** is the same as **dde-poke**, except that conversations are managed automatically. The function issues a poke transaction to set the value of the item specified by **item** to the value specified by **data**. The argument **item** should be a string, or a symbol. If it is a symbol its print name is used.

If **server** names a client service registered with **define-dde-client**, the registered service name is used as the DDE service name. If **server** is any other symbol, the print name of the
symbol is used as the DDE service name. If server is a string, that string is used as the DDE service name.

The topic argument specifies the DDE topic name to be used in the conversation. If it is a symbol, the symbol’s print name is used. If it is a string, the string is used.

For information on the format, type, and errorp arguments, see dde-poke.

If necessary, the function dde-poke* creates a conversation for the duration of the transaction, but if a suitable conversation already exists, the transaction is executed over that conversation. Hence, if several transactions are made with the same service and topic, placing them inside a with-dde-conversation prevents a new conversation being established for each transaction.

If new-conversation-p is set to t, a new conversation is always established for the transaction. This new conversation is always automatically disconnected when the transaction is completed.

If connect-error-p is t (the default value), LispWorks signals an error if a conversation cannot be established. If it is nil, dde-poke* returns nil if a conversation cannot be established. This allows the caller to distinguish between the cases when the server is not running, and when the server is running but the transaction fails.

See also dde-item
dde-request

dde-request Function

Summary Issues a request transaction on a conversation for a specified item.

Package win32
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...
This chapter applies only to LispWorks for Windows

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 success. If successful, result is the return value of the transaction (which may be nil in the case of :string-list), and success is true to indicate success.

On failure, the result of the function depends on the errorp argument. If errorp is t (the default), the function signals an error. If errorp is nil, the function returns (values nil nil).

See also
dde-item
dde-poke
dde-request*

dde-request*

Function

Summary Issues a request transaction on an automatically managed conversation for a specified item.

Package win32

Signature dde-request* service topic item &key format type errorp connect-error-p new-conversation-p => result successp

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>service</td>
<td>A symbol or string.</td>
</tr>
<tr>
<td>topic</td>
<td>A symbol or string.</td>
</tr>
<tr>
<td>item</td>
<td>A string or symbol.</td>
</tr>
<tr>
<td>format</td>
<td>A clipboard format specifier.</td>
</tr>
<tr>
<td>type</td>
<td>A keyword.</td>
</tr>
</tbody>
</table>
The DDE client interface

This chapter applies only to LispWorks for Windows

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 topic argument specifies the DDE topic name to be used in the conversation. If it is a symbol, the symbol’s print name is used. If it is a string, the string is used.

For information on the format, type, and errorp arguments see dde-request.

If necessary, the function dde-request* will create a conversation for the duration of the transaction, but if a suitable conversation already exists, the transaction will be executed over that conversation. Hence, if several transactions will be made with the same service and topic, placing them inside a with-dde-conversation prevents a new conversation being established for each transaction.

If new-conversation-p is set to t a new conversation is always established for the transaction. This new conversation is always automatically disconnected when the transaction is completed.
This chapter applies only to LispWorks for Windows

If `connect-error-p` is `t` (the default value), then LispWorks signals an error if a conversation cannot be established. If it is `nil`, `dde-request` returns `nil` if a conversation cannot be established. This allows the caller to distinguish between the cases when the server is not running, and when the server is running but the transaction fails.

See also
- `dde-item`
- `dde-poke`
- `dde-request`

**define-dde-client**

*Macro*

**Summary**
Registers a client service.

**Package**
`win32`

**Signature**
`define-dde-client name &key service class => name`

**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`

**Signature**

```lisp
(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 `errorp`. If `errorp` is `t` (the default value), then LispWorks signals an error. If `errorp` is `nil`, the body is not executed, and `nil` is returned.

See also `define-dde-client`
This chapter describes the Dynamic Data Exchange (DDE) server interface which is available in the **WIN32** package. You should use this chapter in conjunction with Chapter 18, “Dynamic Data Exchange”.

The **WIN32** package also includes functions for accessing miscellaneous Microsoft Windows functionality, the registry API, and the DDE client interface. These are documented in separate chapters in this manual.

**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**

**Signature**

dde-server-poke server topic item data &key format &allow-other-keys => successp

**Arguments**

- **server**
  - A server object.
- **topic**
  - A topic object.
This chapter applies only to LispWorks for Windows

**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.

**Package**

`win32`

**Signature**

```
dde-server-request server topic item &key format &allow-other-keys => data
```

**Arguments**

`server` A server object.
This chapter applies only to LispWorks for Windows

<table>
<thead>
<tr>
<th>topic</th>
<th>A topic object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>A string.</td>
</tr>
<tr>
<td>format</td>
<td>A keyword.</td>
</tr>
</tbody>
</table>

**Values**

| data | The returned data. |

**Description**

The generic function `dde-server-request` is called in response to a request transaction. A method specializing on the classes of `server` and `topic` should return the data in `item`.

The expected format of the data is given by `format`, which defaults to `:text`. The set of supported formats may be extended in future releases, so applications should always check the value of the format parameter and reject transactions which use formats not supported by the application.

If the request fails, `nil` should be returned.

**See also**

- `dde-poke`
- `dde-request`
- `dde-server-poke`

### dde-server-topic

**Generic Function**

**Summary**

Called whenever a client attempts to connect to a server with a given topic.

**Package**

`win32`

**Signature**

`dde-server-topic server topic-name => topic`

**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.
This chapter applies only to LispWorks for Windows

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 216 for details of the items implemented by this topic.

See also  
dde-topic

dde-topic  

Class

Summary  
The ancestor of all topic classes.

Package  
win32

Superclasses  
standard-object

Subclasses  
dde-system-topic
Description
The class **dde-topic** is the superclass of all topic objects. You can define subclasses using **defclass** and return instances of them by defining a method for the **dde-server-topic** generic function. This allows you to create topics with arbitrary internal state that can be accessed via DDE.

Examples
See **examples\dde\server-dispatching.lisp**

See also
**dde-server-topic**
**dde-system-topic**

---

**dde-topic-items**

**Generic Function**

**Summary**
Returns the valid items in a topic.

**Package**
**win32**

**Signature**
**dde-topic-items** *server topic => item-strings***

**Arguments**
- **server** A server object.
- **topic** A topic object.

**Values**
- **item-strings** A list of strings.

**Description**
The generic function **dde-topic-items** returns a list of strings corresponding to the valid items in the topic. A method specializing on a server and topic should be defined.

If it is not practical to return a list of the items (for example, if the list is potentially infinite), the generic function returns **:unknown**.

**See also**
**dde-server-topic**
**dde-server-topics**
**define-dde-dispatch-topic**

**Macro**

**Summary**  Defines a dispatch topic.

**Package**  win32

**Signature**  

```lisp
define-dde-dispatch-topic name &key server topic-name => name
```

**Arguments**

- **name** A symbol.
- **server** A server class.
- **topic-name** A string.

**Values**

- **name** A symbol.

**Description**  The macro `define-dde-dispatch-topic` defines a dispatching topic. A dispatching topic is a topic which has a fixed name and always exists. Dispatching topics provide dispatching capabilities, whereby appropriate application-supplied code is executed for each supported transaction. Note that the server implementation also provides some dispatching capabilities.

The name of the dispatching topic object is specified by `name`.

The topic is identified by the string `topic-name`.

The class of the server to attach the topic to is given by `server`.

The macro `define-dde-dispatch-topic` returns the name of the dispatching topic, `name`.

Use `define-dde-server-function` with the `:topic` option to define items for a dispatch topic.

**Example**

```lisp
(define-dde-dispatch-topic topic1 :server demo-server)

(define-dde-server-function (item1 :topic topic1)
  :request
  ()
  ..handle topic1.item1 request..)
```
See also  
dde-server-topic  
dde-server-topics  
define-dde-server-function

**define-dde-server**  
*Macro*

**Summary**  
Defines a class for a Lisp DDE server.

**Package**  
win32

**Signature**  
```
define-dde-server class-name service-name => class-name  
define-dde-server class-name superclasses slot-specs options => class-name```

**Arguments**  
- `class-name`: A class name.
- `service-name`: A string.
- `superclasses`: A list of superclasses.
- `slot-specs`: The specifications for the class’ slots.
- `options`: A keyword option.

**Values**  
- `class-name`: A class name.

**Description**  
The macro `define-dde-server` defines a class for a Lisp DDE server. The class inherits from `dde-server`.

The long form of the macro is similar to `defclass`, but with one extra option, `:service`, which is used to specify the service name string to which this server will respond.

The short form is provided to handle the common simple case; `class-name` is the name of the Lisp class to be defined, and `service-name` is the service name string to which this server will respond.
This chapter applies only to LispWorks for Windows

Example

The first example uses the short version of `define-dde-server` to define a class, called `lisp-server`, which has the service name “LISP”.

```
(define-dde-server lisp-server "LISP")
```

The second example shows how to use the long form of the macro to define the same class, and illustrates the use of the `superclasses` and `options` arguments.

```
(define-dde-server lisp-server (dde-server)
  ()
  (:service "LISP"))
```

See also

dde-server-topic
dde-server-topics
dde-topic-items

define-dde-server-function

Macro

Summary

Defines a server function that is called when a specific transaction occurs.

Package

win32

Signature

```
define-dde-server-function name-and-options transaction
  (binding*) form* => name

name-and-options ::= name | (name [[option]])

transaction ::= :request | :poke | :execute

option ::= :server server | :topic-class topic-class | :topic topic | :item item | :format format | :command command | :result-type result-type | :advisep advisep

binding ::= var-binding | execute-arg-binding

var-binding ::= (var :server) | (var :topic) | (var :data [data-type]) | (var :format)

execute-arg-binding ::= var | (var type-spec)
```
Arguments

- **name**: A symbol.
- **transaction**: A keyword.
- **server**: A server object.
- **topic-class**: A topic class.
- **topic**: A symbol naming a dispatch topic.
- **item**: A string.
- **format**: A keyword.
- **command**: A string.
- **result-type**: A data type.
- **advisep**: A boolean.
- **var**: A variable.
- **data-type**: A data type.
- **type-spec**: A data type.
- **form**: A Lisp form.

Values

- **name**: A symbol.

Description

The macro `define-dde-server-function` is used to define a server function, called **name**, which is called when a specific transaction occurs. The defined function may either be attached to a server class (using the dispatching capabilities built into the server implementation) or to a named dispatch topic.

- To attach the definition to a server, **server** should be used to specify the server class. **topic-class** may be used to specify the topic-class for which this definition should be used. It can be a symbol which names a **topic-class**, or t (meaning All topics, this is the default for execute transactions), or **system** (The System topic), or **non-system** (any topic except the System topic). In the case of execute transactions only, **topic-class** defaults
to 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 \texttt{:server} binding causes the variable to be bound to the server object, whereas a \texttt{: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 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, \texttt{dde-execute-command} does not currently generate such strings.

\textbf{See also}\ 
\texttt{dde-execute-command}  
\texttt{define-dde-client}  
\texttt{define-dde-dispatch-topic}  
\texttt{define-dde-server}  

\textbf{start-dde-server} \hspace{1cm} \textit{Function}\hspace{1cm}  
\textbf{Summary}\hspace{1cm} Creates and starts an instance of a DDE server. \hspace{1cm}  
\textbf{Package}\hspace{1cm} \texttt{win32} \hspace{1cm}  
\textbf{Signature}\hspace{1cm} \texttt{start-dde-server name \textasciitilde server} \hspace{1cm}  
\textbf{Arguments}\hspace{1cm} name \hspace{1cm} A DDE server class \hspace{1cm}  
\textbf{Values}\hspace{1cm} server \hspace{1cm} A server object \hspace{1cm}  
\textbf{Description}\hspace{1cm} The function \texttt{start-dde-server} creates an instance of a server of the class specified by \texttt{name} which then starts accepting transactions. If successful the function returns the server, otherwise \texttt{nil} is returned.  
You need to call \texttt{start-dde-server} in a thread that will process Windows messages. This can either be done by using \texttt{capi:execute-with-interface} to run it in the thread of an application's main window (if there is one) or by running it in a dedicated thread as in the example. DDE callbacks will happen in this thread.
Example

(mp:process-run-function
"DDE Server"
()#
(lambda ()
  (win32:start-dde-server 'lispworks-dde-server)
  (loop
    (mp:wait-processing-events nil
    :wait-reason "DDE Request Loop")))))

See also define-dde-server
This chapter describes the C functions available in a LispWorks dynamic library, that is a library created by passing `dll-exports` or `dll-added-files` to `save-image` or `deliver`.

For an overview of this functionality with examples of use, see Chapter 13, “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 306)

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 306.

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 InitLispWorks timed out before LispWorks finished mapping itself from the file.

-3 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 InitLispWorks timed out before after LispWorks mapped itself from the file, but before the initialization was complete.

After InitLispWorks times out, the state of LispWorks can be queried by 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 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 errno number by -1001 - value.

Note: If LispWorks is already initialized or in the process of being initialized, InitLispWorks does not initiate the process of initialization. Therefore the arguments to 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 13, “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:

```c
void __stdcall *LispWorksDlsym (const char * name)
```

On Linux, Macintosh, FreeBSD and x86/x64 Solaris:

```c
void *LispWorksDlsym (const char * name)
```
Description

The C function LispWorksDlsym returns the address of a foreign callable \textit{name} which is defined in Lisp using \texttt{fli:define-foreign-callable}.

LispWorksDlsym first checks if the LispWorks dynamic library finished initializing, and if not uses \texttt{InitLispWorks} to initialize it (with \texttt{MilliTimeOut} 200). If this fails LispWorksDlsym returns NULL. When the LispWorks dynamic library is initialized, LispWorksDlsym returns the address of \textit{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 \texttt{deliver} and \texttt{save-image}. For an overview of LispWorks as a dynamic library, see Chapter 13, “LispWorks as a dynamic library”.

See also

InitLispWorks

LispWorksState

\textbf{C function}

Summary

Returns the state of a LispWorks dynamic library.

Signature

On Windows:

\begin{verbatim}
int __stdcall LispWorksState (int MilliTimeOut)
\end{verbatim}

On Linux, Macintosh, FreeBSD and x86/x64 Solaris:

\begin{verbatim}
int LispWorksState (int MilliTimeOut)
\end{verbatim}

Description

The C function LispWorksState returns the state of a LispWorks dynamic library.

\textit{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 \texttt{QuitLispWorks}, \texttt{LispWorksState} returns -100. Otherwise, it returns the same values as \texttt{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 13, “LispWorks as a dynamic library”.

See also InitLispWorks
QuitLispWorks

SimpleInitLispWorks

Summary
Initializes a LispWorks dynamic library.

Signature
On Windows:

\[
\text{int \_stdcall SimpleInitLispWorks (void)}
\]

On Linux, Macintosh, FreeBSD and x86/x64 Solaris:

\[
\text{int SimpleInitLispWorks (void)}
\]

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 13, “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 if 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 13, “LispWorks as a dynamic library”.

See also deliver
dll-quit
save-image
Symbols

⚠ 139
:< debugger command 15
$ (dollar) variable 26
$$ variable 26
$$ variable 26
(setf fill-pointer) function 172
>: debugger command 15
>, SQL operator 244
? listener command 7
[... ] syntax in Common SQL 243

Numerics

16-bit-string type 670
8-bit-string type 669

A
:a debugger command 20
abort restart 11
accessor-method-slot-definition generic function 150
accessors
dde-item 1243
dde-item 1244
lob-stream-lob-locator 956
socket-stream-socket 368
sql-database-data-error 1016
sql-error-database-message 1016
sql-error-secondary-error-id 1016
storage-exhausted-gen-num 1195
storage-exhausted-size 1195
storage-exhausted-static 1195
storage-exhausted-type 1195
stream-read-timeout 368
stream-writing-timeout 368
action lists 77
defining 78
examples 82
undefining 78
*active-finders* variable 489
adding actions to action lists 80
add-method generic function 151
address space 319
add-special-free-action function 117, 531
add-sql-stream function 251, 911
add-symbol-profiler function 123, 532
adjust-array function 172
advice
after 51
around 51
before 51
example of use 56
facility 49
for macros 53
for methods 53
main chapter 49, 61
removing 53
after advice 51
:after keyword 38
:all debugger command 18
:all keyword 196
all, SQL operator 244
:allocation keyword 42
allocation of stacks 117, 1090
allocation-in-gen-num macro 107, 111,
Index

533
allowing-block-interrupts macro 164, 783
analysing-special-variables-usage
  macro 534
ANSI
  Common Lisp 152, 235
  SQL mode 222, 254
ANSI_QUOTES
  SQL mode 254
any, SQL operator 244
append macro 670
apply-in-pane-process function 174
apply-with-allocation-in-gen
  function 113
apply-with-allocation-in-gen-num
  function 1071
apropos function 387
apropos-list function 388
argument list 715
arguments
  command line 302
  lisp function 715
around advice 51
array-dimension-limit constant 321
array-total-size-limit constant 321
array-weak-p function 537
ASCII 295
at-location macro 490
atomic-decf macro 1072
atomic-exchange macro 1073
atomic-fixnum-decf macro 1074
atomic-fixnum-incf macro 1074
atomic-incf macro 1072
atomicity and thread safety
  in CAPI 174
  in the editor 173
  in the LispWorks implementation 172
atomic-pop macro 1075
atomic-push macro 1075
attach-ssl function 277
attach-ssl function 283, 345
attribute-type function 238, 912
augmented-string type 290, 1076
augmented-string-p function 1076
avoid-gc function 112, 538

B
:break debugger command 14
backtrace 13
quick backtrace 14
verbose backtrace 14
:backtrace keyword 39
barrier-arriver-count function 785
barrier-change-count function 785
barrier-count function 786
barrier-disable function 188, 787
barrier-enable function 188, 787
barrier-name function 788
barrier-pass-through function 788
barrier-unblock function 789
barrier-wait function 187, 790
base slot 238
base-char type 289, 671
base-character type 670
base-character-p function 671
base-char-code-limit constant 672
base-char-p function 671
base-string type 290, 389
base-string-p function 672
:base-table class option 934
before advice 51
:before keyword 37
*binary-file-type* variable 396
*binary-file-types* variable 393, 396
:bindings keyword 24
binds-who function 539
block-promotion macro 539
BOM 297, 299
Bordeaux threads
  APIs needed for 843
:bq debugger command 15
:break keyword 39
break-new-instances-on-access
  function 323
break-on-access function 324
:break-on-exit keyword 39
break-on-unresolved-functions
  function 663
browser 673
*browser-location* variable 673
browsing documentation 673
buffered-stream class 1037
:bug-form listener command 7
building-universal-intermediate-p
  function 541
Byte Order Mark 297, 299

C
:ce debugger command 20
C functions
dlopen 143
dlsym 143
GetProcAddress 143
Index

InitLispWorks 146, 307, 1271
LispWorksDlsym 1274
LispWorksState 146, 1275
LoadLibrary 143
malloc 106
memalign 106
QuitLispWorks 147, 1277
realloc 106
SimpleInitLispWorks 1276
cache-table-queries function 234, 914
*cache-table-queries-default* variable 915
call Unix functions from Lisp 311
calling AppleScript 1078
call-next-advice function 52, 673
call-next-advice macro 59
calls-who function 541
call-system function 1077
call-system-showing-output function 1079
canonicalize-dspec function 491
catch frame, examining 12
catchers keyword 24
:caused-by keyword 196
:cc debugger command 18
cd macro 542
cdr-assoc function 1081
close-registry-key function 314, 1211
close-serial-port function 903
Cocoa application 811
Cocoa application bundle saving 131
Cocoa event loop 811
code signing in saved image 610
coerce function 390
coerce-to-gesture-spec function 1083
*coff-loading-verbose* 668
collect-generation-2 function 109, 111, 547
collect-highest-generation function 111, 547
collect-registry-subkeys function 315, 1212
collect-registry-values function 315, 1213
command line 302
command line arguments 1142
-build 302
-display 303
-env 303
-environment 303
-eval 303
-IIOPHost 303
-IIOPNumeric 303
-init 303
-load 304
-lw-no-redirection 304
-multiprocessing 304
-no-restart-function 304
-ORBPort 304
--relocate-image 304
--reserve-size 305
-siteinit 305
command line processing 1142
commands
listener 1092
top level 1092
commit function 233, 235, 261, 915
Common Lisp
systems. See system
Common SQL
[... ] syntax 243
case of names 253
database classes 223
database connection 225
database encoding 254
date fields 249, 257
encoding 226
errors 251
Functional DDL 237
Functional DML 231
functional interface 230
I/O recording 251
initialization 223
iteration 242
main chapter 219
Object Oriented DDL 239
Object Oriented DML 241
object-oriented interface 238
ODBC compliance 221
programmatic interface 247
result types 231, 232, 256
supported databases 221
symbolic syntax 243
transaction handling 226, 234, 256
utilities 249
Common SQL errors
sql-connection-error 252
sql-database-data-error 252
sql-database-error 251
sql-fatal-error 252
sql-temporary-error 252
sql-timeout-error 252
sql-user-error 251
compare-and-swap macro 1085
compilation-speed 88
compile function 391
compile-file function 392
compile-file-if-needed function 549
compiler
comparison with interpreter 85
control 88
levels of safety 89
main chapter 85
optimization of 88–92
workings of 87
compiler explanations 94, 398
compiler help 94, 398
*compiler-break-on-error*
variable 548
compile-system function 194, 675
compute-applicable-methods-using-classes generic function 151
compute-class-potential-initargs
generic function 326
compute-discriminating-function
generic function 151, 328
concatenate function 397
concatenate-system function 677
Condition variables 186
conditions
external-format-error 526
file-encoding-resolution-error 1102
sql-connection-error 1015
sql-database-data-error 1015
sql-database-error 1016
sql-fatal-error 1018
sql-temporary-error 1024
sql-timeout-error 1024
sql-user-error 1024
condition-variable-broadcast
function 187, 794
condition-variable-signal
function 187, 794
condition-variable-wait function 187, 795
condition-variable-wait-count
function 187, 796
configuring the printer 140
connect function 221, 225, 226, 227, 253, 916
connected-databases function 225, 923
*connect-if-exists* variable 923
connecting to a database
MySQL 227
ODBC 226
Oracle 225
PostgreSQL 229
console 606
console application 606
constants
array-dimension-limit 321
array-total-size-limit 321
base-char-code-limit 672
most-positive-fixnum 320
continue restart 11
copy-preferences-from-older-version function 1086
copy-to-weak-simple-vector function 118, 550
count-gen-num-allocation function 115, 1088
Counting semaphores 188
create-index function 238, 924
create-macos-application-bundle function 551
create-registry-key function 314, 315, 1215
create-simple-process function 797
create-table function 238, 925
create-universal-binary function 553
create-view function 238, 926
create-view-from-class function 239, 928
creation of process 162
:ctx-configure-callback initarg 368
current frame 15
current process 162
current-pathname function 679
*current-process* variable 162, 163, 799
current-process-block-interrupts function 164, 800
current-process-in-cleanup-p function 801
current-process-pause function 801
current-process-unblock-interrupts function 164, 803
current-stack-length function 555
customization
main chapter 130
of editor 136
:ev inspector command 27

d
:di inspector command 27
database
classes in Common SQL 223
connection in Common SQL 225
encoding in Common SQL 254
table names 253
database-name function 225, 928
databases supported 221
dates
in Common SQL 249, 257
dde-advertise-client-data generic function 212
dde-advertise-start function 212, 1227
dde-advertise-start* function 212, 1230
dde-advertise-stop function 213, 1231
dde-advertise-stop* function 213, 1232
dde-client-advertise-data generic function 1234
dde-connect function 211
dde-connect macro 1234
dde-disconnect function 211
dde-disconnect macro 1235
dde-execute function 1236
dde-execute* function 1237
dde-execute-command* function 214
dde-execute-command function 214, 1237
dde-execute-command* function 1238
dde-execute-string function 214, 1240
dde-execute-string* function 214, 1241
dde-item accessor 1243
dde-item* accessor 1244
dde-item function 213
dde-poke function 213, 1246
dde-poke* function 1248
dde-request function 213, 1249
dde-request* function 1251
dde-server-poke generic function 215, 1257
dde-server-request generic function 215, 1258
dde-server-topic generic function 1259
dde-server-topics generic function 216, 1260
dde-system-topic class 1261
dde-topic class 1261
dde-topic-items generic function 1262
DDL 237, 239
debug 88
debugger
commands 13
Index

control variables 22
invoking from the tracer 39
main chapter 9
debugger commands
:< 15
:> 15
:a 20
call 18
:b 14
:bb 15
:c 20
:ce 18
:ed 18
:error 18
:func 19
:1 17
:lambda 18
:lf 19
:n 16
:p 15
:res 20
:ret 20
:top 20
:v 16
*debug-initialization-errors-in-
  snapshot* variable 1088
*debug-io* variable 22
debug-other-process function 804
*debug-print-length* variable 23, 467
*debug-print-level* variable 23, 468
declare macro 93, 397
declaration
:alias 398
:explain 398
:invisible-frame 398
:lambda-list 398
:special-dynamic 98, 398
:special-fast-access 98, 398
:special-global 98, 398
:values 398
declare :explain 94, 398
declare special form 88, 92, 398
decode-external-string function 524
defmacro 492
defadvice macro 50, 54, 59, 681
default directory 582
default file directory 582
*default-action-list-sort-time*
  variable 684, 81
*default-character-element-type*
  parameter 684
*default-character-element-type*
  variable 290, 292, 293, 294, 298
*default-database* variable 223, 225, 229
*default-database-type* variable 224, 229
default-eol-style function 1089
*default-libraries* variable 667
*default-package-use-list*
  variable 555
:default-pathname keyword 195
*default-process-priority*
  variable 805
*default-profiler-collapse*
  variable 556
*default-profiler-cutoff*
  variable 556
*default-profiler-limit* variable 557
*default-profiler-sort* variable 557
*default-simple-process-priority*
  variable 805
*default-stack-group-list-length*
  variable 117, 1090
*default-update-objects-max-len*
  variable 930
defclass macro 403
defglobal-parameter macro 558
defglobal-variable macro 558
define-action macro 80, 683
define-action-list macro 78, 687
define-atomic-modify-macro
  macro 1090
define-dde-client function 212
define-dde-client macro 1253
define-dde-dispatch-topic macro 216, 1263
define-dde-server macro 214, 1264
define-dde-server-function
  macro 215, 1265
define-dspec-alias macro 493
define-dspec-class macro 494
define-foreign-callable macro 191
define-form-parser macro 73, 497
define-top-loop-command macro 1092
definition specs 45
defpackage macro 407
defparameter macro 558
defparser
  error handling with 204
defparser macro 201, 899
  functions defined by 203
defstruct macro 140
*defstruct-generates-print-object-
Index

method* variable 140
defsystem macro 195, 195–199, 689
examples of use 197
*defsystem-verbose* variable 694
deftransform macro 463
defvar macro 559
def-view-class macro 220, 238, 239, 930
delete-advice macro 53, 59, 559
delete-directory function 694
delete-instance-records generic function 241, 937, 950
delete-records function 233, 234, 938
delete-registry-key function 314, 1216
delete-sql-stream function 251, 939
deliver function 143, 302, 307, 695
deliverable
  filename 302, 725
  pathname 302, 725
delivering a DLL 143
delivering a dynamic library 143
describe function 25, 409
*describe-length* variable 696, 27
*describe-level* variable 696
*describe-level* special variable 26
describe-object generic function 26
*describe-print-length* variable 698, 26
*describe-print-level* variable 698, 26
destroy-ssl function 286, 347
destroy-ssl-ctx function 286, 347
detach-ssl function 285, 348
detect-eol-style function 1094
detect-japanese-encoding-in-file function 1095
detect-unicode-bom function 1096
diagnostic utilities
  for action lists 81
  :direction initarg 368, 956, 1037
directory function 409
*directory-link-transparency* special variable 410
*directory-link-transparency* variable 1097
disable-sql-reader-syntax function 249, 940
*disable-trace* variable 560
disassemble function 414
discard-source-info function 507
disconnect function 225, 940
dismiss-splash-screen function 1203
DLL 143
  filename 302, 725
  pathname 302, 725
dll-quit function 147, 699
dlopen C function 143
dlsym C function 143
:dm inspector command 27
DML 231, 241
DNS 350
documentation generic function 415
$(dollar) variable 26
$$ variable 26
$$$ variable 26
domain 350
do-nothing function 702
:dont-know keyword 17
do-profiling function 123, 561
do-query macro 235, 260, 941
do-rand-seed function 349
DOS command
call-system 1077
call-system-showing-output 1079
open-pipe 1158
dotted-list-length function 701
dotted-list-p function 702
double-float type 416
:dr inspector command 27
drop-index function 238, 942
drop-table function 238, 943
drop-view function 238, 944
drop-view-from-class function 239, 944
dspec-class function 501
*dspec-classes* variable 501
dspec-defined-p function 502
dspec-definition-locations function 502
dspec-equal function 503
dspec-name function 504
dspec-primary-name function 504
dspec-progenitor function 505
dspecs
  aggregate 68
  canonical 62
  displaying definitions 73
  examples 61
  finding definitions 72
  grouping definitions 67
  new defining forms 66
  parts 68
  recording definitions 71
dspec-subclass-p function 506
dspec-undefiner function 506
dump-form function 563
dump-forms-to-file function 564

1285
Index

dylib 143
dynamic libraries 143, 306
dynamic library 143
  memory clash 146
  relocation 146

E
:ed debugger command 18
editor
  customizing 136
editor source code 138
:ef-spec initarg 1102
:element-type initarg 368, 1038
Emacs 3
enable-sql-reader-syntax function 231, 249, 945
encode-lisp-string function 525
encoding
  changing default for files 298
enlarge-generation function 110, 112, 565
enlarge-static function 566
ensure-loads-after-loads function 1097
ensure-memory-after-loads function 1098
ensure-process-cleanup function 805
ensure-ssl function 286
ensure-ss1 function 349
ensure-stores-after-memory function 1099
ensure-stores-after-stores function 1099
*enter-debugger-directly* variable 703
:entrycond keyword 40
enum-registry-value function 315, 1217
environment-variable function 703
esql-specializer class 151
esql-specializer-object function 151
errno-value function 705
:eval debugger command 18
error handlers
  in applications 474
error handling
  in parser generator 204
error output 1152
errors in Common SQL 251
EUC-JP 295
:eval-after keyword 38
:eval-before keyword 38
evaluating
  forms during tracing 37–39
example-compile-file function 706
example-file function 705
example-load-binary-file function 707
except, SQL operator 244
exception handlers
  in applications 474
exception handling
  for action lists 80
exceptions
  handling 474
executable 302
  filename 302, 725
  pathname 302, 725
executable-log-file function 469
execute-actions macro 707
execute-command function 237, 946
execute-with-interface function 174
execution functions 77
execution profiling 121
execution stack
  examining 12
:exitcond keyword 40
expand-generation-1 function 111, 567
extend-current-stack function 568
extended-char type 709
extended-character type 709
extended-character-p function 710
extended-char-p function 710
*extended-spaces* variable 777, 1100
extended-time function 569
extended-time macro 115, 116, 127
external format
  changing default for files 298
  external format specification 295
External formats 295
external programs
  calling from Lisp 311
external-format-error condition 526
external-format-foreign-type function 526
*external-formats* variable 711
external-format-type function 527
:extra-initargs class option 325, 327, 404

F
false function 712
fasl (fast load)
  description 85
FDDL 237
FDML 241
*features* variable 416
file-directory-p function 712
*file-encoding-detection-algorithm* variable 298, 1101
file-encoding-resolution-error condition 1102
*file-eol-style-detection-algorithm* variable 298, 1102
filename of deliverable 302, 725
filename of DLL 302, 725
filename of dynamic library 725
filename of executable 302, 725
filename of lisp image 302, 725
*filename-pattern-encoding-matches* variable 1103
files
  load-on-demand 139
file-stat-blocks function 1114
file-stat-device function 1113
file-stat-device-type function 1114
file-stat-group-id function 1113
file-stat-inode function 1113
file-stat-last-access function 1114
file-stat-last-change function 1114
file-stat-last-modify function 1114
file-stat-links function 1114
file-stat-mode function 1114
file-stat-owner-id function 1113
file-stat-size function 1114
file-string function 571
file-writable-p function 572
find-database function 225, 946
find-depspec-locations function 507
find-encoding-option function 1103
find-external-char function 527
find-filename-pattern-encoding-match function 1104
find-name-locations function 508
find-object-size function 111, 572
find-process-from-name function 806
find-regexp-in-string function 713
finish-heavy-allocation function 573
fixnum type 319
fixnum-safety 88
flag-not-special-free-action function 117, 574
flag-special-free-action function 117, 575
FLI types
  lpcstr 1207
  lpcwchar 1208
lpstr 1207
lpcwchar 1208
lpwstr 1209
p-oci-env 1002
p-oci-file 1002
p-oci-lob-locator 1002
p-oci-lob-or-file 1002
p-oci-svc-ctx 1003
ssl-cipher-pointer 375
ssl-cipher-pointer-stack 375
ssl-ctx-pointer 376
ssl-pointer 378
str 1207
tstr 1208
wstr 1209
float 88
foreign callbacks 191
foreign types
  p-oci-env 264, 972
  p-oci-file 264, 990
  p-oci-lob-locator 264, 990
  p-oci-svc-ctx 264, 996
  ssl-cipher-pointer 279
  ssl-ctx-pointer 279
  ssl-pointer 279
foreign-slot-value function 100
foreign-symbol-address function 664
forms
  evaluating when tracing 37–39
frame, examining 12
  :free-lob-locator-on-close initarg 956
  ;fun debugger command 19
funcallable-standard-class class 151
funcallable-standard-instance-access function 150
funcallable-standard-object class 152, 328
function, altering with advice 49
Functional DDL 237
Functional DML 231
functional interface in Common SQL 230
function-lambda-list function 715
functions
  add-special-free-action 117, 531
  add-sql-stream 251, 911
  add-symbol-profiler 123, 532
  apply-with-allocation-in-gen 113
  apply-with-allocation-in-gen-num 1071
  apropos 387

1287
delete-registry-key 314, 1216
delete-sql-stream 251, 939
deliver 143, 302, 307, 695
describe 25, 409
destroy-ssl 286, 347
destroy-ssl-ctx 286, 347
detach-ssl 285, 348
detect-eol-style 1094
detect-japanese-encoding-in-file 1095
detect-unicode-bom 1096
directory 409
disable-sql-reader-syntax 249, 940
disassemble 414
discard-source-info 507
disconnect 225, 940
dismiss-splash-screen 1203
dll-quit 147, 699
do-nothing 702
do-profiling 123, 561
do-rand-seed 349
dotted-list-length 701
dotted-list-p 702
drop-index 238, 942
drop-table 238, 943
drop-view 238, 944
drop-view-from-class 239, 944
dspec-class 501
dspec-defined-p 502
dspec-definition-locations 502
dspec-equal 503
dspec-name 504
dspec-primary-name 504
dspec-progenitor 505
dspec-subclass-p 506
dspec-undefined 506
dump-form 563
dump-forms-to-file 564
dump-registry-number 315, 1217
dump-registry-value 315, 1217
dump-sql-registry-value 315, 1217
dump-sql-stream 251, 939
dis(gen)-num-segments-fragmentation
Index

state 115, 1105
 gensym 116
 gesture-spec-data 1108
 gesture-spec-modifiers 1109
 gesture-spec-p 1110
 gesture-spec-to-character 1112
 get-current-process 162, 808
 get-default-generation 111, 579
 get-file-stat 1112
 get-folder-path 316, 1114
 get-foreign-symbol 665
 get-gc-parameters 112, 580
 get-host-entry 350
 get-process 163, 808
 get-process-private-property 809
 get-serial-port-state 903
 get-socket-address 352
 get-socket-peer-address 352
 get-temp-directory 581
 get-verification-mode 353
 get-working-directory 582
 inspect 26
 int32* 1120
 int32+ 95, 1121
 int32- 95, 1122
 int32/ 1124
 int32/= 1125
 int32< 1125
 int32<= 1126
 int32<= 1127
 int32= 1127
 int32> 1128
 int32>> 1129
 int32>>= 1129
 int32-1+ 1123
 int32-1- 1124
 int32-aref 1130
 int32-logand 1135
 int32-lognor 1135
 int32-lognot 1136
 int32-logorcl 1137
 int32-logorcl 1137
 int32-logtest 1138
 int32-logxor 1139
 int32-minusp 1139
 int32-plusp 1140
 int32-to-integer 1140
 integer-to-int32 1142
 interactive-stream-p 420
 intern-eql-specializer 151
 ip-address-string 354
 last-callback-on-thread 812
 lisp-image-name 302, 725
 lisp-name-to-foreign-name 666
 list-all-processes 163, 813
 list-attributes 238, 952
 list-attribute-types 238, 951
 list-classes 241, 953
 list-sql-streams 251, 954
 list-tables 238, 955
 load-all-patches 727
 load-data-file 1143
 load-logical-pathname-translations 421
 load-system 728
 lob-stream-lob-locator 263
 local-dspec-p 510
 locale-file-encoding 1144
 locally-disable-sql-reader-syntax 249, 957
 locally-enable-sql-reader-syntax 249, 957
 lock-locked-p 814
 lock-name 180, 816
 lock-owned-by-current-process-p 815
 lock-owner 180, 817
 lock-recursively-locked-p 816
 lock-recursive-p 815
 log-bug-form 471
 logs-directory 473
 long-namestring 302, 1205, 1207
 long-site-name 301, 422
 low-level-atomic-place-p 1145
 mailbox-empty-p 818
 mailbox-peek 819
 mailbox-read 820
 mailbox-reader-process 821

1290
Index

print-pretty-gesture-spec 1163
print-profile-list 126, 592
print-query 232, 1003
process-alive-p 835
process-all-events 836
process-allow-scheduling 162, 836
process-arrest-reasons 837
process-break 164, 838
process-continue 838
process-exclusive-lock 838
process-exclusive-unlock 839
process-idle-time 840
process-interrupt 164, 165, 842
process-join 843
process-kill 164, 843
process-lock 180, 844
process-mailbox 845
process-name 163, 846
process-p 846
process-plist 190, 846
process-poke 847
process-priority 163, 848
process-private-property 848
process-property 190, 849
process-reset 850
process-run-function 162, 851
process-run-reasons 853
process-run-time 854
process-send 855
process-sharing-lock 856
process-sharing-unlock 857
process-stop 170, 857
process-stopped 170
process-stopped-p 858
process-unlock 859
process-unstop 170, 860
process-wait 164, 186, 861
process-wait-for-event 862
process-wait-function 164, 862
process-wait-local 164, 863
process-wait-local-with-periodic-checks 865
process-wait-local-with-timeout 867
process-wait-local-with-timeout-and-periodic-checks 868
process-wait-with-timeout 164, 183, 186, 869
process-whostate 870
proclaim 88, 93, 434
product-registry-path 314, 1165
profiler-tree-from-function 598
profiler-tree-to-function 599
ps 163, 872
pushnew-to-process-private-property 190, 871
pushnew-to-process-property 190, 871
query 237, 259, 1004
query-registry-key-info 315, 1220
query-registry-value 315, 1221
quit 4, 312, 739
read-dhparams 360
read-dhparms 278
read-foreign-modules 667
read-serial-port-char 904
read-serial-port-string 905
reconnect 225, 1005
record-definition 514
references-who 600
regexp-find-symbols 741
registry-key-exists-p 315, 1222
registry-value 315, 1223
remove-advice 53, 59, 742
remove-from-process-private-property 190, 873
remove-from-process-property 190, 873
remove-process-private-property 190, 874
remove-process-property 190, 875
remove-special-free-action 117, 601
remove-symbol-profiler 123, 601
reset-profiler 123, 602
restore-sql-reader-syntactic-state 249, 1007
results for traced 38
rollback 233, 235, 261, 1007
room 111, 112, 115, 306, 436
room-values 306, 1166
round-to-single-precision 745
run-shell-command 1167
safe-locale-file-encoding 1172
save-argument-real-p 603
save-current-session 604
save-image 131, 143, 302, 307, 312, 605
save-image-with-bundle 613
save-tags-database 517
save-universal-from-script 313, 615
sbchar 746
schedule-timer 189, 876
schedule-timer-milliseconds 878
schedule-timer-relative 879
schedule-timer-relative-
Index

milliseconds 881
select 231, 241, 1008
semaphore-acquire 188, 882
semaphore-count 188, 883
semaphore-name 188, 884
semaphore-release 188, 884
semaphore-wait-count 188, 885
serial-port 904
serial-port-input-available-p 906
set-application-themed 1206
set-array-single-thread-p 616
set-array-weak 118, 617
set-automatic-gc-callback 115, 1173
set-blocking-gen-num 116, 1174
set-debugger-options 24
set-default-character-element-type 746
set-default-generation 107, 111, 618
set-default-segment-size 116, 1177
set-delay-promotion 116, 1178
setf cdr-assoc 1081
setf timer-name 886
set-file-dates 1179
set-gc-parameters 105, 108, 112, 1180
set-gen-num-gc-threshold 116, 1180
set-hash-table-weak 118, 621
set-make-instance-argument-checking 334
set-maximum-memory 111, 1181
set-maximum-segment-size 114, 116, 1183
set-memory-check 1184
set-memory-exhausted-callback 1185
set-minimum-free-space 108, 111, 623
set-process-profiling 123, 124, 624
set-profiler-threshold 123, 626
set-promotion-count 627
set-registry-value 315, 1224
set-serial-port-state 907
set-signal-handler 1187
set-spare-keeping-policy 116, 1189
set-ssl-ctx-dh 278, 364
set-ssl-ctx-options 278, 365
set-ssl-ctx-password-callback 278, 366
set-ssl-library-path 288, 367
set-who 633
set-system-message-log 628
setup-atomic-funcall 1190
setup-for-alien-threads 191
set-up-profiler 122, 629
set-verification-mode 362
short-namestring 302
short-site-name 301, 441
simple-augmented-string-p 1192
simple-base-string-p 747
simple-char-p 748
simple-process-p 886
simple-text-string-p 749
single-form-form-parser 517
single-form-with-options-form-parser 518
socket-stream-address 373
socket-stream-ctx 278, 373
socket-stream-peer-address 374
socket-stream-ssl 278, 374
software-type 301, 442
software-version 301, 443
source-debugging-on-p 633
split-sequence 750
split-sequence-if 751
split-sequence-if-not 752
sql 248, 1014
sql-expression 248, 1017
sql-operation 247, 1019
sql-operator 248, 1021
sql-recording-p 251, 1022
sql-stream 251, 1023
ssl-add-client-ca 279
ssl-addr-chain-cert 279
ssl-cipher-get-bits 280
ssl-cipher-get-name 280
ssl-cipher-get-version 280
ssl-clear-num-renegotiations 280
ssl-ctrl 280
ssl-ctx-addr-client-ca 280
ssl-ctx-addr-extra-chain-cert 280
ssl-ctx-ctrl 280
ssl-ctx-get-max-cert-list 280
ssl-ctx-get-mode 280
ssl-ctx-get-options 280
ssl-ctx-get-read-ahead 280
ssl-ctx-verify-policy 280
ssl-ctx-load-verify-locations 280
ssl-ctx-load-verify-locations 280
ssl-ctx-load-verify-policy 280
ssl-ctx-sess-get-cache-size 280
ssl-ctx-sess-get-cache-size 280
ssl-ctx-sess-get-cache-size 280
ssl-ctx-sess-get-cache-size 280
ssl-ctx-sess-get-cache-size 280
ssl-ctx-set-client-ca-list 280
ssl-ctx-set-max-cert-list 280
ssl-ctx-set-mode 280
ssl-ctx-set-options 280
ssl-ctx-set-read-ahead 281
ssl-ctx-set-temp-dh 281

1293
ssl-ctx-set-tmp-rsa 281
ssl-ctx-use-certificate-chain-file 281
ssl-ctx-use-certificate-file 281
ssl-ctx-use-privatekey-file 281
ssl-ctx-use-rsaprivatekey-file 281
ssl-get-current-cipher 281
ssl-get-max-cert-list 281
ssl-get-mode 281
ssl-get-options 281
ssl-get-verify-mode 281
ssl-get-version 281
ssl-load-client-ca-file 281
ssl-need-tmp-rsa 281
ssl-new 286, 377
ssl-num-renegotiations 281
ssl-session-reused 281
ssl-set-accept-state 281, 284
ssl-set-client-ca-list 281
ssl-set-connect-state 281, 284
ssl-set-max-cert-list 281
ssl-set-mode 281
ssl-set-options 282
ssl-set-tmp-dh 282
ssl-set-tmp-rsa 282
ssl-total-renegotiations 282
ssl-use-certificate-file 282
ssl-use-privatekey-file 282
ssl-use-rsaprivatekey-file 282
standard-instance-access 150
start-dde-server 1269
start-profiling 123, 124, 634
start-sql-recording 251, 1025
start-tty-listener 753
start-up-server 379
start-up-server-and-mp 383
statcip 1194
status 225, 1026
stchar 753
stop-profiling 123, 124, 636
stop-sql-recording 251, 1027
string-append 754
string-ip-address 384
sweep-all-objects 117, 637
sweep-gen-num-objects 1195
switch-static-allocation 105, 106, 638
symeval-in-process 170, 886
table-exists-p 1028
text-string-p 756
timer-expired-p 887
timer-name 888
toggle-source-debugging 101, 639
total-allocation 111, 640
traceable-dspec-p 519
trace-new-instances-on-access 337
trace-on-access 338
tracing inside 43
tracing-enabled-p 520
tracing-state 521
true 756
true-name 458
try-compact-in-generation 110, 112, 650
try-move-in-generation 110, 112, 651
typed-aref 1196
unbreak-new-instances-on-access 341
unbreak-on-access 342
unicode-alpha-char-p 300, 758
unicode-alphanumeric-p 300, 759
unicode-both-case-p 300, 759
unicode-char-equal 300, 760
unicode-char-greaterp 300, 761
unicode-char-not-equal 300, 763
unicode-char-not-greaterp 300, 763
unicode-char-not-lessp 300, 764
unicode-lower-case-p 300, 765
unicode-string-equal 300, 766
unicode-string-greaterp 300, 767
unicode-string-not-equal 300, 768
unicode-string-not-lessp 300, 769
unicode-string-not-greaterp 300, 770
unicode-string-not-lessp 300, 771
unicode-upper-case-p 300, 772
unnotice-fd 890
unschedule-timer 890
untrace-new-instances-on-access 342
untrace-on-access 343
update-instance-for-different-class 460
update-instance-for-redefined-class 460
update-objects-joins 1029
update-records 233, 234, 260, 1031
user-homedir-pathname 315
user-preference 314, 773
valid-external-format-p 528
vector-pop 174
vector-push 174
vector-push-extend 174
Index

wait-for-input-streams 1197
wait-for-input-streams-returning-first 1199
wait-processing-events 891
wait-serial-port-state 907
whitespace-char-p 777
who-binds 655
who-calls 656
who-references 657
who-sets 657
without-preemption 165
with-output-to-fasl-file 659
write-serial-port-char 908
write-serial-port-string 909
yield 897
fundamental-binary-input-stream
class 1039
fundamental-binary-output-stream
class 1040
fundamental-binary-stream
class 1040
fundamental-character-input-stream
class 270, 1041
fundamental-character-output-stream
class 270, 1042
fundamental-input-stream
class 1043
fundamental-output-stream
class 1044
fundamental-stream
class 1044

G

garbage collection, see also storage management
main chapter 103
GBK 296
gc-generation function 112, 115, 116, 575
gc-if-needed function 112, 579
general-handle-event generic function 807
generation definition 104
generation 2 109
generation-number function 110, 1106
generic functions
accessor-method-slot-definition 150
add-method 151
class-extra-initargs 325
close 389
compute-applicable-methods-using-classes 151
compute-class-potential-initargs 326
compute-discriminating-function 151, 328
dde-client-advice-data 1234
dde-server-poke 1257
dde-server-request 1258
dde-server-topic 1259
dde-server-topics 1260
dde-topic-items 1262
delete-instance-records 241, 937, 950
describe-object 26
documentation 415
general-handle-event 807
get-inspector-values 716
input-stream-p 419
instance-refreshed 241
make-instance 428
make-method-lambda 150
open-stream-p 433
output-stream-p 433
print-object 140
process-a-class-option 329
process-a-slot-option 331
slot-boundp-using-class 150, 335
slot-makunbound-using-class 150, 336
slot-value-using-class 150, 336
stream-advance-to-column 1045
stream-check-eof-no-hang 1046
stream-clear-input 272, 1046
stream-clear-output 273, 1047
stream-element-type 270, 447
stream-file-position 1048
stream-fill-buffer 1048
stream-finish-output 273, 1049
stream-flush-buffer 1050
stream-force-output 273, 1051
stream-fresh-line 1051
stream-line-column 273, 1052
stream-listen 272, 1053
stream-output-width 1054
stream-peek-char 1054
stream-read-buffer 1055
stream-read-byte 1056
stream-read-char 271, 1057
stream-read-char-no-hang 1057
stream-read-line 1058
stream-read-sequence 1059
stream-read-timeout 1060
stream-start-line-p 273, 1060
stream-terpri 1061
stream-unread-char 271, 1062
stream-write-buffer 1062
stream-write-byte 1063
stream-write-char 272, 1064
stream-write-sequence 1064
stream-write-string 1065
update-instance-from-records 241, 1028
update-record-from-instance 1032
update-record-from-slot 241, 1032
update-records-from-instance 241
update-slot-from-record 241, 1033
:gen-num-initarg 1198
:gen-num-segments-fragmentation-state function 115, 1105
gensym function 116
gesture-spec-accelerator-bit variable 1107
gesture-spec-control-bit variable 1107
gesture-spec-data function 1108
gesture-spec-hyper-bit variable 1108
gesture-spec-meta-bit variable 1109
gesture-spec-modifiers function 1109
gesture-spec-p function 1110
gesture-spec-shift-bit variable 1111
gesture-spec-super-bit variable 1111
gesture-spec-to-character function 1112
:get listener command 6
current-process function 162, 808
default-generation function 111, 579
file-stat function 1112
folder-path function 316, 1114
foreign-symbol function 665
form-parser function 509
go-parameters function 112, 580
host-entry function 350
inspector-values generic function 716
GetProcAddress C function 143
process function 163, 808
process-private-property function 809
serial-port-state function 903
socket-address function 352
socket-peer-address function 352
temp-directory function 581
unix-error function 718
user-profile-directory function 317, 1116
verification-mode function 353
working-directory function 582
grammar
non-terminal 202
resolving ambiguities 203
graphics ports xxxvii
> SQL operator 244
*grep-command* variable 718
*grep-command-format* variable 719
*grep-fixed-args* variable 720
guess-external-format function 297, 1117
GUI application 606
H
:h inspector command 27
*handle-existing-action-in-action-list* variable 720, 80
*handle-existing-action-list* variable 721, 80
*handle-existing-defpackage* variable 582
*handle-missing-action-in-action-list* variable 722
*handle-missing-action-in-action-list* variable 81
*handle-missing-action-list* variable 721, 81
*handle-old-in-package* variable 584
*handle-old-in-package-used-as-make-package* variable 584
handler frame, examining 12
:handler keyword 24
*handle-warn-on-redefinition* variable 138, 722
hardcopy-system function 723
hash tables
weak 425
heap size 319
:help listener command 7
:hidden keyword 24
:hidden-packages* variable 23, 470
:his inspector command 7
hook functions 77
host 350
host name 350
hostname 350
I
:i inspector command 27
i8n 289
image
saving 131
Index

image size 304
incf macro 173
init file 724
*init-file-name* variable 724
initialization
  of Common SQL 223
initialization file 724
initialize-database-type
  function 223
initialize-database-type function 947
*initialized-database-types* variable 224, 948
initialize-multiprocessing
  function 168, 810
*initial-processes* variable 147, 162, 168, 811
InitLispWorks C function 146, 307, 1271
input-stream-p generic function 419
insert-records function 233, 260, 949
:inside keyword 24
inspect function 26
inspector
  main chapter 25
  REPL 25
  teletype 25
inspector commands
  :cv 27
  :d 27
  :dm 27
  :dr 27
  :i 27
  :m 27, 28
  :s 27
  :sh 27
  :u 27
  :ud 27, 28
*inspect-print-length* variable 27
*inspect-print-level* variable 27
*inspect-through-gui* variable 724
instance-refreshed generic function 241
in-static-area macro 106, 1118
int32* function 1120
int32* function 95, 1121
int32- function 95, 1122
int32=> function 1119
int32=> function 1124
int32/= function 1125
int32=< function 1125
int32<= function 1126
int32<= function 1127
int32= function 1127
int32=> function 1128
int32-> function 1129
int32-> function 1129
+int32-0+ symbol macro 1122
int32-1+ function 1123
int32-1- function 1124
+int32-1+ symbol macro 1123
int32-aref function 1130
int32-logand function 1131
int32-logandc1 function 1131
int32-logandc2 function 1132
int32-logbitp function 1133
int32-logeqv function 1133
int32-logior function 1134
int32-logmand function 1135
int32-lognor function 1135
int32-lognot function 1136
int32-logorc1 function 1137
int32-logorc2 function 1137
int32-logtest function 1138
int32-logxor function 1139
int32-minusp function 1139
int32-plusp function 1140
int32-to-integer function 1140
int32-zerop function 1141
integer-to-int32 function 1142
interactive-stream-p function 420
interface
  between parser generator and lexical analyser 204
Common SQL initialization 223
Internationalization 289
intern-sql-specializer function 151
interpreter
  differences from compiler 85
interruptable 88
intersect, SQL operator 244
invalid superclass 153
:invisible keyword 24
IP Address 350
ip-address-string function 354
ISO8859-1 295

J
JIS 295
join slot 239

K
keywords
  :after 38
Index

:all 196
:allocation 42
:backtrace 39
:before 37
:bindings 24
:break 39
:break-on-exit 39
:catchers 24
:caused-by 196
:default-pathname 195
:don’t-know 17
:entrycond 40
:eval-after 38
:eval-before 38
:exitcond 40
:handler 24
:hidden 24
:inside 43
:invisible 24
:maximum-buffer-size 106
:maximum-overflow 109
:members 195
:minimum-buffer-size 106
:minimum-for-sweep 108, 109
:minimum-overflow 109
:new-generation-size 109
:non-symbol 24
:package 195
:previous 197
:process 42
:requires 197
:restarts 24
:rules 196
:source-only 196
:step 40
:trace-output 41
:when 42

L
:l debugger command 17
:lambda debugger command 18
:last-callback-on-thread function 812
:Latin-1 295
:*latin-1-code-pages* variable 1204
:levels of safety, see compiler
:slot debugger command 19
:library formats 667
:lightweight processes 161
:*line-arguments-list* variable 302, 1142
:lisp image
:filename 302, 725

pathname 302, 725
:lisp-image-name function 302, 725
:lisp-name-to-foreign-name function 666
:LispWorks
:customizing 130
:lightweight processes in 161
:processes 161
:quitting 4, 84
:saving 2
:starting 1, 84
:LispWorks as a DLL 143
:LispWorks as a dynamic library 143
:LispWorks as a shared library 143
:*liseworks-directory* variable 726
:LispWorksLsym C function 1274
:LispWorksState C function 146, 1275
:list-all-processes function 163, 813
:list-attributes function 238, 952
:list-attribute-types function 238, 951
:list-classes function 241, 953
:listener 753
:main chapter 5
:top level commands 1092
:listener commands
:? 7
:bug-form 7
:get 6
:help 7
:his 7
:redo 6
:use 7
:listener process 812
:listener prompt 738
:list-sql-streams function 251, 954
:list-tables function 238, 955
:load-all-patches function 727
:load-data-file function 1143
:*load-fasl-or-lisp-file* variable 585
:LoadLibrary C function 143
:load-logical-pathname-translations function 421
:load-on-demand 139
:*load-source-if-newer* 729
:load-system function 728
:lob-locator initarg 956
:lob-stream class 259, 263, 268, 955
:lob-stream-lob-locator accessor 956
:lob-stream-lob-locator function 263
:local-dspec-p function 510
:locale-file-encoding function 1144
:locally-disable-sql-reader-syntax
function 249, 957
locally-enable-sql-reader-syntax function 249, 957
location macro 511
lock-locked-p function 814
lock-name function 180, 816
lock-owned-by-current-process-p function 815
lock-owner function 180, 817
lock-recursively-locked-p function 816
lock-recursive-p function 815
locks 179
log-bug-form function 471
logs-directory function 473
long-float type 421
long-namestring function 302, 1205, 1207
long-site-name function 301, 422
loop macro 230, 235, 422, 958
extensions in Common SQL 242
loop, extensions in Common SQL 236
Low level atomic operations 175
low-level-atomic-place-p function 1145
lpcstr FLI type 1207
lpcstr FLI type 1208
lpwstr FLI type 1209
lpwstr FLI type 1208
lpwstr FLI type 1209
M
:inspector command 27, 28
Mach-O bundle 610
Mach-O dynamically linked shared library 608
macros
advice 53
allocation-in-gen-num 107, 111, 533
allowing-block-interrupts 164, 783
analysing-special-variables-usage 534
appendf 670
at-location 490
atomic-def 1072
atomic-exchange 1073
atomic-fixnum-def 1074
atomic-fixnum-inf 1074
atomic-inf 1072
atomic-pop 1075
atomic-push 1075
block-promotion 539
call-next-advice 59
cd 542
compare-and-swap 1085
dde-connect 1234
dde-disconnect 1235
declare 93, 397
def 492
defadvice 50, 54, 59, 681
defclass 403
defglobal-parameter 558
defglobal-variable 558
define-action 80, 685
define-action-list 78, 687
define-atomic-modify-macro 1090
define-dde-client 1253
define-dde-dispatch-topic 1263
define-dde-server 1264
define-dde-server-function 1265
define-dspec-alias 493
define-dspec-class 494
define-foreign-callable 191
define-form-parser 497
define-top-loop-command 1092
defpackage 407
defparameter 558
defparser 201, 203, 899
defstruct 140
defsystem 195, 195–199, 689
deftransform 463
defvar 559
def-view-class 238, 239, 930
delete-advice 53, 59, 559
do-query 235, 260, 941
execute-actions 707
extended-time 115, 116, 127
in-static-area 106, 1118
location 511
loop 422, 958
profile 123, 596
rebinding 740
removef 744
restart-case 436
simple-do-query 235, 260, 1012
step 444
time 449
trace 451
undefine-action 80, 757
undefine-action-list 79, 757
untrace 459
unwind-protect-blocking-interrupts 164, 653
unwind-protect-blocking-inter-
rupts-in-cleanups 164, 654
when-let 775
when-let* 776
with-action-item-error-handling 777
with-action-list-mapping 779
with-dde-conversation 1254
with-debugger-wrapper 484
with-exclusive-lock 180, 892
with-hash-table-locked 658
with-heavy-allocation 112, 659
with-interrupts-blocked 164, 893
with-lock 180, 894
with-modification-change 1199
with-modification-check-macro 1200
with-noticed-socket-stream 385
with-other-threads-disabled 165, 1201
without-interrupts 165, 896
without-preemption 165, 897
with-output-to-string 461
with-registry-key 314, 1225
with-sharing-lock 180, 895
with-stream-input-buffer 1066
with-stream-output-buffer 1068
with-transaction 233, 234, 1034
with-unique-names 780
mailbox-empty-p function 818
mailbox-peek function 819
mailbox-read function 820
mailbox-reader-process function 821
mailbox-send function 821
mailbox-wait-for-event function 822
*main-process* variable 824
make-array function 118, 174, 424
make-barrier function 188, 824
make-condition-variable function 825
make-gesture-spec function 1146
make-hash-table function 118, 1225
make-instance generic function 428
make-lock function 826
make-mailbox function 828
make-method-lambda generic function 150
make-mt-random-state function 731
make-named-timer function 829
make-semaphore function 188, 830
make-sequence function 429
make-simple-int32-vector function 1151
make-ssl-ctx function 283, 284, 354
make-stderr-stream function 1152
make-symbol function 116
make-timer function 189, 831
make-typed-aref-vector function 1152
make-unregistered-action-list function 730
malloc C function 106
map function 429
map-all-processes function 832
map-all-processes-backtrace function 832
map-process-backtrace function 833
map-processes function 834
map-query function 235, 260, 960
mark
and sweep 107
mark-and-sweep function 105, 112, 586
marking-ge function 115, 116, 1153
:maximum-buffer-size keyword 106
*maximum-ordinary-windows* variable 137
:maximum-overflow keyword 109
*max-trace-indent* variable 45, 588
memalign C function 106
:members keyword 195
memory allocation during tracing 42
avoiding 306
memory management 304
garbage collection strategy 108
image reduction 116
mark and sweep 107
overflow 109
timing in 115, 116
memory-growth-margin function 111, 1155
merge function 430
merge-of-specs function 1155
Mersenne Twister 732
Metaobject Protocol 149
metaobject protocol
class options 330
slot options 332
method
advice 53
method-combination class 151
methods
tracing 45
:minimum-buffer-size keyword 106
:minimum-for-sweep keyword 108, 109
:minimum-overflow keyword 109
minus, SQL operator 244
mod 2^32 arithmetic 95
modify-hash function 172, 173, 589
modifying a database 233
MOP
  AMOP compatibility 149
  class options 330
  slot options 332
most-positive-fixnum constant 320
mt-random function 732
mt-random-state type 733
*mt-random-state* variable 733
mt-random-state-p function 734
*multibyte-code-page-ef* variable 1205
multi-processing
  locks 179
MySQL
  connecting 227
  MySQL client library 228
  Mac OS X 228
  *mysql-library-directories* variable 228, 229, 961
  *mysql-library-path* variable 228, 229, 962
N
  :n debugger command 16
name-defined-dspecs function 511
name-definition-locations function 512
name-only-form-parser function 513
New in LispWorks 6.0
  allowing-block-interrupts 783
  analysing-special-variables-usage 534
  array-weak-p 537
  atomic-decf 1072
  atomic-exchange 1073
  atomic-fixnum-decf 1074
  atomic-fixnum-incf 1074
  atomic-incf 1072
  atomic-pop 1075
  atomic-push 1075
  barrier-arriver-count 785
  barrier-change-count 785
  barrier-count 786
  barrier-disable 787
  barrier-enable 787
  barrier-name 788
  barrier-pass-through 788
  barrier-unblock 789
  barrier-wait 790
  Blocking interrupts 164
  choose-unicode-string-hash function 675
compare-and-swap 1085
condition-variable-broadcast 794
condition-variable-signal 794
condition-variable-wait 795
condition-variable-wait-count 796
create-macos-application-bundle 551
current-process-block-interrupts 800
current-process-in-cleanup-p 801
current-process-pause 801
current-process-unblock-interrupts 803
defglobal-parameter 558
defglobal-variable 558
define-atomic-modify-macro 1090
do-profiling 561
ensure-loads-after-loads 1097
ensure-memory-after-store 1098
ensure-stores-after-memory 1099
execute-log-file 469
:func 19
generation-number 1106
get-process-private-property 809
get-temp-directory 581
last-callback-on-thread 812
lock-locked-p 816
lock-owned-by-current-process-p 815
lock-recursively-locked-p 816
lock-recursive-p 815
log-bug-form 471
logs-directory 473
low-level-atomic-place-p 1145
make-barrier 824
make-semaphore 830
modify-hash 589
new-function 825
process-all-events 836
process-exclusive-lock 838
process-exclusive-unlock 839
process-poke 847
process-private-property 848
process-property 849
process-sharing-lock 856
process-sharing-unlock 857
process-wait-local 863
process-wait-local-with-periodic-checks 865
process-wait-local-with-timeout 867
process-wait-local-with-timeout-and-periodic-checks 868
pushnew-to-process-private-property 871
recursive locks 826
remove-from-process-private-property 873
remove-from-process-private-property 873
remove-process-private-property 874
remove-process-property 875
save-current-session 604
save-image-with-bundle 613
semaphore-acquire 882
semaphore-count 883
semaphore-name 884
semaphore-release 884
set-array-single-thread-p 615
set-system-message-log 628
setup-atomic-funcall 1190
sharing locks 826
short-namestring 1205, 1207
split-sequence 750
split-sequence-if 751
split-sequence-if-not 752
unicode-alpha-char-p 300, 758
unicode-alphanumericp 300, 759
unicode-both-case-p 300, 759
unicode-char-equal 300, 760
unicode-char-greaterp 300, 761
unicode-char-lessp 300, 762
unicode-char-not-equal 300, 763
unicode-char-not-greaterp 300, 763
unicode-char-not-lessp 300, 764
unicode-lower-case-p 300, 765
unicode-string-equal 300, 766
unicode-string-greaterp 300, 767
unicode-string-lessp 300, 768
unicode-string-not-equal 300, 769
unicode-string-not-greaterp 300, 770
unicode-string-not-lessp 300, 771
unicode-upper-case-p 300, 772
unwind-protect-blocking-interrupts 653
unwind-protect-blocking-interrupts-in-cleanups 654
with-exclusive-lock 892
with-hash-table-locked 658
with-interrupts-blocked 893
with-modification-change 1199
with-modification-check-macro 1200
with-other-threads-disabled 1201
with-sharing-lock 895
:new-generation-size keyword 109
:non-symbol keyword 24
non-terminal in grammar 202
normal-gc function 112, 590
not, SQL operator 244
notice-fd function 835

O
object
object-oriented interface in Common SQL 238
static 106
object finalization 117
Object Oriented DDL in Common SQL 239
Object Oriented DML in Common SQL 241
object-address function 1156
ODBC
connecting 226
OODDL 239
OODML 241
open function 431
opening a URL 1160
open-pipe function 1157
open-registry-key function 314, 1219
open-serial-port function 901
OpenSSL 277
openssl-version function 358
open-stream-p generic function 433
open-tcp-stream function 277, 283, 355
open-url function 1160
operating system 301
optimization
fast 32-bit arithmetic 95
floating point 96
foreign slot access 100
of compiler 88
tail call 97
optimization declarations 88
optimization hints 94, 398
optimize 88
optimize qualities 89–91
:optimize-slot-access class option 150,
153, 337, 403
Oracle
connecting 225
Oracle Call Interface
in Common SQL 226
ora-lob-append function 267, 963
ora-lob-assign function 266, 964
ora-lob-char-set-form function 265, 964
ora-lob-char-set-id function 965
ora-lob-close function 267, 966
ora-lob-copy function 267, 967
ora-lob-create-empty function 260, 266, 968
ora-lob-create-temporary function 268, 969
ora-lob-disable-buffering function 268, 970
ora-lob-element-type function 265, 971
ora-lob-enable-buffering function 268, 971
ora-lob-env-handle function 263, 972
ora-lob-erase function 267, 973
ora-lob-file-close function 267, 974
ora-lob-file-close-all function 267, 975
ora-lob-file-exists function 975
ora-lob-file-get-name function 976
ora-lob-file-is-open function 977
ora-lob-file-open function 267, 978
ora-lob-file-set-name function 267, 978
ora-lob-flush-buffer function 268, 979
ora-lob-free function 266, 980
ora-lob-free-temporary function 268, 981
ora-lob-get-buffer function 263, 268, 982
ora-lob-get-chunk-size function 266, 984
ora-lob-get-length function 266, 985
ora-lob-internal-lob-p function 265, 266, 985
ora-lob-is-equal function 266, 986
ora-lob-is-open function 266, 987
ora-lob-is-temporary function 266, 987
ora-lob-load-from-file function 267, 988
ora-lob-lob-locator function 263, 989
ora-lob-locator-is-init function 266, 990
ora-lob-open function 267, 991
ora-lob-read-buffer function 265, 268, 992
ora-lob-read-foreign-buffer function 263, 265, 268, 995
ora-lob-read-into-plain-file function 268, 994
ora-lob-svc-ctx-handle function 263, 996
ora-lob-trim function 267, 997
ora-lob-write-buffer function 265, 268, 998, 1000
ora-lob-write-foreign-buffer function 263, 265, 268
ora-lob-write-from-plain-file function 268
ora-lob-write-from-plain-file function 999
output
  trace 41
output-backtrace function 474
output-stream-p generic function 433
P
:p debugger command 15
package
  hiding 23
:package keyword 195
packages
  allocation of 116
*packages-for-warn-on-redefinition*
  variable 138, 591
parameters
  command line 302
*default-character-element-type* 684
parse-float function 591
parse-form-dspec function 514
parser generator main chapter 201
parser, error handling 204
passing runtime parameters 302
patches
  saving an image with 4
path
  long form on Windows 302
  short form on Windows 302
pathname of deliverable 302, 725
pathname of DLL 302, 725
pathname of dynamic library 725
pathname of executable 302, 725
pathname-location function 734
pem-read function 278
pem-read function 359
pid-exit-status function 1161
pipe
  open 1158
PL/SQL 946
platform 301
*features* 416
software-type 442
software-version 443
p-oci-env FLI type 1002
p-oci-env foreign type 264, 972
p-oci-file FLI type 1002
p-oci-file foreign type 264, 990
p-oci-lob-locator FLI type 1002
p-oci-lob-locator foreign type 264, 990
p-oci-lob-or-file FLI type 1002
p-oci-svc-ctx FLI type 1003
p-oci-svc-ctx foreign type 264, 996
pointer-from-address function 1162
pointers weak 617
PostgreSQL
connecting 229
PostScript Printer Description files 140
PPD files 140–141
precompile-regexp function 735
:previous keyword 197
print-action-lists function 736
print-actions function 82, 736
*print-binding-frames* variable 23, 475
*print-catch-frames* variable 23, 477
*print-command* variable 737
printer
process-allow-scheduling function 162, 836
process-break function 164, 838
process-continue function 838
processes allocation of 117
process-exclusive-lock function 838
process-exclusive-unlock function 839
process-idle-time function 840
*process-initial-bindings* variable 163, 841
process-interrupt function 164, 165, 842
process-join function 843
process-kill function 164, 843
process-lock function 180, 844
process-mailbox function 845
process-name function 163, 846
process-p function 846
process-plist function 190, 846
process-poke function 847
process-priority function 163, 848
process-private-property function 848
process-property function 190, 849
process-reset function 850
process-run-function function 162, 851
process-run-reasons function 853
process-run-time function 854
process-send function 855
process-sharing-lock function 856
process-sharing-unlock function 857
process-stop function 170, 857
process-stopped function 170
process-stopped-p function 858
process-unlock function 859
process-unstop function 170, 860
process-wait function 164, 186, 861
process-wait-for-event function 862
process-wait-function function 164, 862
process-wait-local function 164, 863
process-wait-local-with-periodic-checks function 865
process-wait-local-with-timeout function 867
process-wait-local-with-timeout-and-periodic-checks

1304
### Index

**Index**

- **function 868**
  - process-wait-with-timeout function 164, 183, 186, 869
  - process-whostate function 870
  - proclaim function 88, 93, 434
  - product-registry-path function 314, 1165

- **profile macro 123, 596**
- **profile time 122**
- **profiler**
  - interpretation of results 126
  - main chapter 121
  - pitfalls 126
  - setting up 122
  - *profiler-print-out-all* variable 595
  - *profiler-threshold* variable 597
  - profiler-tree-from-function function 598
  - profiler-tree-to-function function 599
  - *profile-symbol-list* variable 123, 598

- **profiling**
  - execution 121
  - program 121
  - program profiling 121
  - promotion 107
  - *prompt* variable 738
  - prompt in listener 738
  - *prompt* variable 8
  - ps function 163, 872

- **pseudo operators**
  - sql-boolean-operator 247
  - sql-function 247
  - sql-operator 247
  - push macro 173
  - pushnew-to-process-private-property function 190, 871
  - pushnew-to-process-property function 190, 871

- **Q**
  - :q inspector command 27
  - query function 237, 259, 1004
  - query-registry-key-info function 315, 1220
  - query-registry-value function 315, 1221
  - quick backtrace 14
  - quit function 4, 312, 739
  - QuitLispWorks C function 147, 1277

- **R**
  - raw 32-bit arithmetic 95
  - read-dhparams function 360
  - read-dhparms function 278
  - read-eval-print loop 5, 753
  - read-foreign-modules function 667
  - read-serial-port-char function 904
  - read-serial-port-string function 905
  - :read-timeout initarg 368
  - real time 123
  - realloc C function 106
  - rebind macro 740
  - reconnect function 225, 1005
  - record-definition function 514
  - *record-source-files* variable 516
  - *redreinition-action* variable 138, 516
  - redo 139
  - :redo listener command 6
  - requires-who function 600
  - regexp 713, 741
  - regexp-find-symbols function 741
  - registry
    - API on Windows 314, 1211
    - registry-key-exists-p function 315, 1222
    - registry-value function 315, 1223
    - regular expression 713, 741
    - regular expression matching 695, 713, 741
    - relocating 306
    - relocation 304
  - remove-advice function 53, 59, 742
  - removef macro 744
  - remove-from-process-private-property function 190, 873
  - remove-from-process-property function 190, 873
  - remove-process-private-property function 190, 874
  - remove-process-property function 190, 875
  - remove-special-free-action function 117, 601
  - remove-symbol-profiler function 123, 601
  - removing actions from action lists 80
  - REPL 5, 753
  - REPL inspector 25
  - :requires keyword 197
  - *require-verbose* variable 744
  - :res debugger command 20
  - reserved words 591
  - reset-profiler function 123, 602

---

1305
Index

restart 11
restart frame, examining 12
restart-case macro 436
:restarts keyword 24
restore-sql-reader-syntax-state function 249, 1007
:ret debugger command 20
rollback function 233, 235, 261, 1007
room function 111, 112, 115, 306, 436
room-values function 306, 1166
round-to-single-precision function 745
:rules keyword 196
run-shell-command function 1167
runtime parameters 302

S
:s inspector command 27
safe-locale-file-encoding function 1172
safety 88
save-argument-real-p function 603
save-current-session function 604
save-image function 131, 143, 302, 307, 312, 605
save-image-with-bundle function 613
save-tags-database function 517
save-universal-from-script function 313, 615
saving images 131
sbchar accessor 292
sbchar function 746
schar accessor 292
schedule-timer function 189, 876
schedule-timer-milliseconds function 878
schedule-timer-relative function 879
schedule-timer-relative-milliseconds function 881
scheduling of processes 163
segmentation violation in compiled code 91
select function 231, 241, 1008
select, SQL operator 244
semaphore-acquire function 188, 882
semaphore-count function 188, 883
semaphore-name function 188, 884
semaphore-release function 188, 884
semaphore-wait-count function 188, 885
serial-port function 904
serial-port-input-available-p function 906
set-application-themed function 1206
set-array-single-thread-p function 616
set-array-weak function 118, 617
set-automatic-gc-callback function 115, 1173
set-blocking-gen-num function 116, 1174
set-debugger-options function 24
set-default-character-element-type function 746
set-default-generation function 107, 111, 618
set-default-segment-size function 116, 1177
set-delay-promotion function 116, 1178
setf cdr-assoc function 1081
setf timer-name function 886
set-file-dates function 1179
set-gen-num-gc-threshold function 116, 1180
set-hash-table-weak function 118, 621
set-make-instance-argument-checking function 334
set-maximum-memory function 111, 1181
set-memory-check function 1184
set-memory-exhausted-callback function 1185
set-minimum-free-space function 108, 111, 623
set-process-profiling function 123, 124, 624
set-profiler-threshold function 123, 626
set-promotion-count function 627
set-registry-value function 315, 1224
set-maximum-segment-size function 114, 1183
set-memory-violation function 907
set-signal-handler function 1187
set-spare-keeping-policy function 116, 1189
set-ssl-ctx-dh function 278
set-ssl-ctx-dh function 364
set-ssl-ctx-options function 278
set-ssl-ctx-options function 365
set-ssl-ctx-password-callback function 278
set-ssl-ctx-password-callback function 366
set-ssl-library-path function 288, 367

1306
Index

sets-who function 633
set-system-message-log function 628
setup-atomic-funcall function 1190
setup-for-alien-threads function 191
set-up-profiler function 122, 629
set-verification-mode function 362
*sg-default-size* variable 1191
:sh inspector command 27
shared libraries 143, 306
shared library 143
shared object file 143
Shift JIS 295
short-float type 321, 440
short-namestring function 302
short-site-name function 301, 441
Show Paths From Editor command 101
shutdown 84
simple-augmented-string type 291, 1192
simple-augmented-string-p function 1192
simple-base-string type 291, 292, 441
simple-base-string-p function 747
simple-char type 289, 748
simple-char-p function 748
simple-do-query macro 235, 260, 1012
SimpleInitLispWorks C function 1276
simple-int32-vector type 96, 1193
simple-process-p function 886
simple-string type 291, 292
simple-text-string type 291, 292, 749
simple-text-string-p function 749
single-float type 321, 441
single-form-form-parser function 517
single-form-with-options-form-parser function 518
single-threaded arrays 174
hash tables 174
:size initarg 1195
SLIME 3
slot-bound-using-class generic function 150, 335
slot-makunbound-using-class generic function 150, 336
slot-value
atomic operations 175
slot-value-using-class generic function 150, 336
:socket initarg 368
socket-error class 367
socket-stream class 277
socket-stream class 283, 368
socket-stream-address function 373
socket-stream-ctx function 278
socket-stream-ctx function 373
socket-stream-peer-address function 374
socket-stream-socket accessor 368
socket-stream-ssl function 278
socket-stream-ssl function 374
software-type function 301, 442
software-version function 301, 443
some SQL operator 244
source-debugging-on-p function 633
*source-found-action* variable 136
:source-only keyword 196
space 88
special actions 117
special forms
declare 88, 92, 398
special variables
*describe-level* 26
*directory-link-transparency* 410
speed 88
splash screen 1203
split-sequence function 750
split-sequence-if function 751
split-sequence-if-not function 752
SQL
database functions 247
database operators 247
direct specification 237
mode 254
stored procedure 237, 946
sql function 248, 1014
SQL pseudo operators
sql-boolean-operator 247, 1019
sql-function 247, 1019
sql-operator 247, 1019
sql-boolean-operator SQL pseudo operator 247
sql-boolean-operator SQL pseudo operator 247, 1019
sql-connection-error condition 1015
sql-connection-error error 252
sql-database-data-error accessor 1016
sql-database-data-error condition 1015
sql-database-data-error error 252
sql-database-error condition 1016
sql-database-error error 251
*sql-enlarge-static* variable 1017
sql-error-database-message accessor 1016
sql-error-secondary-error-id
accessor 1016
sql-expression function 248, 1017
sql-fatal-error condition 1018
sql-fatal-error error 252
sql-function pseudo operator 247
sql-function SQL pseudo operator 247, 1019
*sql-libraries* variable 224, 1018
*sql-loading-verbose* variable 224, 234, 1019
sql-operation function 247, 1019
sql-operator function 248, 1021
sql-operator pseudo operator 247
sql-operator SQL pseudo operator 247, 1019
sql-recording-p function 251, 1022
sql-stream function 251, 1023
sql-temporary-error condition 1024
sql-temporary-error error 252
sql-timeout-error condition 1024
sql-timeout-error error 252
sql-user-error condition 1024
sql-user-error error 251
square bracket syntax 243
ssl-add-client-ca function 279
ssl-cipher-get-bits function 279
ssl-cipher-get-name function 280
ssl-cipher-get-version function 280
ssl-cipher-pointer FLI type 375
ssl-cipher-pointer foreign type 279
ssl-cipher-pointer FLI type 375
ssl-clear-num-renegotiations function 280
ssl-closed class 288, 376
ssl-condition class 288, 376
:ssl-configure-callback initarg 368
ssl-ctrl function 280
:ssl-ctx initarg 368
ssl-ctx-add-client-ca function 280
ssl-ctx-add-extra-chain-cert function 280
ssl-ctx-ctrl function 280
ssl-ctx-get-max-cert-list function 280
ssl-ctx-get-mode function 280
ssl-ctx-get-opts function 280
ssl-ctx-get-read-ahead function 280
ssl-ctx-get-verify-mode function 280
ssl-ctx-load-verify-locations function 280
ssl-ctx-need-tmp-rsa function 280
ssl-ctx-pointer FLI type 376
ssl-ctx-pointer foreign type 279
ssl-ctx-sess-get-cache-mode function 280
ssl-ctx-sess-get-cache-size function 280
ssl-ctx-sess-set-cache-mode function 280
ssl-ctx-sess-set-cache-size function 280
ssl-ctx-sess-set-cache-size function 280
ssl-ctx-set-client-ca-list function 280
ssl-ctx-set-max-cert-list function 280
ssl-ctx-set-mode function 280
ssl-ctx-set-options function 280
ssl-ctx-set-read-ahead function 281
ssl-ctx-set-tmp-dh function 281
ssl-ctx-set-tmp-rsa function 281
ssl-ctx-use-certificate-chain-file function 281
ssl-ctx-use-certificate-file function 281
ssl-ctx-use-privatekey-file function 281
ssl-ctx-use-reprivatekey-file function 281
ssl-error class 288, 377
ssl-failure class 288, 377
ssl-get-current-cipher function 281
ssl-get-max-cert-list function 281
ssl-get-mode function 281
ssl-get-options function 281
ssl-get-verify-mode function 281
ssl-get-version function 281
ssl-load-client-ca-file function 281
ssl-need-tmp-rsa function 281
ssl-new function 286, 377
ssl-num-renegotiations function 281
ssl-pointer FLI type 378
ssl-pointer foreign type 279
ssl-session-reused function 281
ssl-set-accept-state function 281, 284
ssl-set-client-ca-list function 281
ssl-set-connect-state function 281, 284
ssl-set-max-cert-list function 281
ssl-set-mode function 281
ssl-set-options function 282
ssl-set-tmp-dh function 282
ssl-set-tmp-rsa function 282
:ssl-side initarg 368
ssl-total-renegotiations function 282
ssl-use-certificate-file function 282
ssl-use-privatekey-file function 282
ssl-x509-lookup class 288, 378
stack
  examining 12
  extension 568
  size 555, 1191
stack size 117, 1090
*stack-overflow-behaviour*
  variable 1193
stacks
  allocation 117
standard-accessor-method class 150
standard-class class 151
standard-db-object class 238, 1025
standard-instance-access function 150
standard-object class 172
standard-reader-method class 150
standard-writer-method class 150
start 84
start LispWorks 1
start-dde-server function 215, 1269
starting LispWorks 1, 84
start-profiling function 123, 124, 634
start-sql-recording function 251, 1025
start-tty-listener function 753
startup 84
startup image 1203
startup relocation 306
startup screen 1203
startup window 1203
start-up-server function 379
start-up-server-and-mp function 383
:static initarg 1195
static object
  allocation in memory management 106
staticp function 1194
status function 225, 1026
stchar accessor 292
stchar function 753
stderr 1152
:step keyword 40
step macro 444
*step-compiled* variable 446
*step-filter* variable 447
stepper, entering when tracing 40
*step-print-env* variable 447
stop-profiling function 123, 124, 636
stop-sql-recording function 251, 1027
storage management
  main chapter 103
storage-exhausted-gen-num accessor 1195
storage-exhausted-size accessor 1195
storage-exhausted-static accessor 1195
storage-exhausted-type accessor 1195
str FLI type 1207
:stream initarg 368
stream-advance-to-column generic
  function 1045
stream-check-eof-no-hang generic
  function 1046
stream-clear-input generic
  function 272, 1046
stream-clear-output generic
  function 273, 1047
stream-element-type generic
  function 270, 447
stream-file-position generic
  function 1048
stream-fill-buffer generic
  function 1048
stream-finish-output generic
  function 273, 1049
stream-flush-buffer generic
  function 1050
stream-force-output generic
  function 273, 1051
stream-fresh-line generic function 1051
stream-line-column generic
  function 273, 1052
stream-listen generic function 272, 1053
stream-output-width generic
  function 1054
stream-peek-char generic function 1054
stream-read-buffer generic
  function 1055
stream-read-byte generic function 1056
stream-read-char generic function 271, 1057
stream-read-char-no-hang generic
  function 1057
stream-read-line generic function 1058
stream-read-sequence generic
  function 1059
stream-read-timeout accessor 368
stream-read-timeout generic
  function 1060
streams
Index

defining new 270
directionality 270
example 270–275
input 271
instantiating 274
output 272
user defined 269
stream-start-line-p generic function 273, 1060
stream-terpri generic function 1061
stream-unread-char generic function 271, 1062
stream-write-buffer generic function 1062
stream-write-byte generic function 1063
stream-write-char generic function 272, 1064
stream-write-sequence generic function 1065
stream-write-timeout generic function 1066
string construction 292
string type 290, 291, 448
string types 290
string-append function 754
string-ip-address function 384
superclass
invalid 153
sweep 107
sweep-all-objects function 117, 637
sweep-gen-num-objects function 1195
switch-static-allocation function 105, 106, 638
symbol macros
+int12-0+ 1122
+int12-1+ 1123
*symbol-alloc-gen-num* variable 111, 116, 638
symbolic query syntax 243
symbolic syntax in Common SQL 243
symbols
allocation of 116
syneval-in-process function 170, 886
Synchronization barriers 187
syntax, in Common SQL 243
system
compile 675
defining 194–199
introduction to 193–194
load 728
members of 195
plan 196
print 723
rules 196–197
system commands
running directly 1078
running via a shell 1078
T
table-exists-p function 1028
tail call 97
tail call merging 97
tail call optimization 97
tail merge 97
tail recursion 97
tail-call 97
teleotype inspector 25
*terminal-debugger-block-multiprocessing* variable 481
text-string type 290, 755
text-string-p function 756
threads
allocation of 117
time macro 449
timer-expired-p function 887
timer-name function 888
timers 189
I/O 189
input and output 189
multiprocessing 189
process 189
threading issues 189
toggle-source-debugging function 101, 639
tools
inspector 25
:top debugger command 20
top-level loop 5
total-allocation function 111, 640
trace
main chapter 35
trace macro 451
traceable-dspec-p function 519
*traced-arglist* variable 37, 38, 40, 46, 641
*traced-results* variable 38, 46, 452, 642
*trace-indent-width* variable 46, 643
*trace-level* variable 46, 644
trace-new-instances-on-access function 337
trace-on-access function 338
:trace-output keyword 41
Index

*trace-output* variable 46
*trace-print-circle* variable 46, 645
*trace-print-length* variable 26, 46, 646
*trace-print-level* variable 26, 46, 647
*trace-print-pretty* variable 47, 648

tracer
 :after option 38
 :allocation option 42
 :before option 37
 :break option 39
 :break-on-exit option 39
 commands available 37–44
 definition specs 45
 directing output 41
 entering the stepper 40
 :entrycond option 40
 :eval-after option 38
 :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 42
 methods, tracing 45
 :process option 42
 restricting to a process 42
 :step option 40
 traced function, arguments for 37
 traced functions, results for 38
 :trace-output option 41
 *trace-verbose* variable 649
 tracing-enabled-p function 520
 tracing-state function 521
 transaction handling
 in Common SQL 226, 234, 256
 true function 756
 truename function 458
 try-compact-in-generation function 110, 112, 650
 try-move-in-generation function 110, 112, 651
 tstr FLI type 1208
 tty 606
 :type initarg 1195
 typed-aref function 1196
 types
 16-bit-string 670
 8-bit-string 669
 augmented-string 290, 1076
 base-char 289, 671
 base-character 670
 base-string 290, 389
 character 289
 double-float 416
 extended-char 709
 extended-character 709
 fixnum 319
 int32 95, 1119
 long-float 421
 mt-random-state 733
 short-float 321, 440
 simple-augmented-string 291, 1192
 simple-base-string 291, 292, 441
 simple-char 289, 748
 simple-int32-vector 96, 1193
 simple-string 291, 292
 simple-text-string 291, 292, 749
 single-float 321, 441
 string 290, 291, 448
 text-string 290, 755

U
 :u inspector command 27
 UCS-2 295
 :ud inspector command 27, 28
 unbreak-new-instances-on-access function 341
 unbreak-on-access function 342
 undefine-action macro 80, 757
 undefine-action-list macro 79, 757
 Unicode 289
 unicode-alpha-char-p function 300, 758
 unicode-alphanumericp function 300, 759
 unicode-both-case-p function 300, 760
 unicode-char-equal function 300, 761
 unicode-char-lessp function 300, 762
 unicode-char-not-equal function 300, 763
 unicode-char-not-greaterp function 300, 764
 unicode-char-not-lessp function 300, 765
 unicode-string-equal function 300, 766
 unicode-string-greaterp function 300, 767
 unicode-string-lessp function 300, 768
Index

unicode-string-not-equal function 300, 769
unicode-string-not-greaterp function 300, 770
unicode-string-not-lessp function 300, 771
unicode-upper-case-p function 300, 772
union, SQL operator 244
universal binaries 312
helper functions 541, 603
saving
  advanced 553
  simply 615
UNIX command
call-system 1077
call-system-showing-output 1079
open-pipe 1158
run-shell-command 1168
Unix commands
calling from Lisp 311
Unix functions
calling from Lisp 311
unnotice-fd function 890
*unresolved-messages* 668
unschedule-timer function 890
untrace macro 499
untrace-new-instances-on-access function 342
untrace-on-access function 343
unwind-protect-blocking-interrupts macro 164, 653
unwind-protect-blocking-interrupts-in-cleanups macro 164, 654
update-instance-for-different-class function 460
update-instance-for-redefined-class function 460
update-instance-from-records generic function 241, 1028
update-objects-joins function 1029
update-record-from-slot generic function 241, 1032
update-records function 233, 234, 260, 1031
update-records-from-instance generic function 241, 1032
update-slot-from-record generic function 241, 1033
URL
  opening 1160
:use listener command 7
user defined stream 269
user-homedir-pathname function 315
user-preference function 314, 773
UTF-$ 293, 298
utilities in Common SQL 249

V
d:v debugger command 16
validate-superclass generic function 151, 152
valid-external-format-p function 528
variables
$(dollar) 26
$$ 26
$$$ 26
*active-finders* 489
*binary-file-type* 396
*binary-file-types* 393, 396
*browser-location* 673
*cache-table-queries-default* 915
*check-network-server* 1083
*compiler-break-on-error* 548
*connect-if-exists* 923
*current-process* 162, 163, 799
*debug-initialization-errors-in-snap-shot* 1088
*debug-io* 22
*debug-print-length* 23, 467
*debug-print-level* 23, 468
*default-action-list-sort-time* 81, 684
*default-character-element-type* 290, 292, 293, 294, 298
*default-database* 223, 225, 929
*default-database-type* 224, 929
*default-libraries* 667
*default-package-use-list* 555
*default-process-priority* 805
*default-profiler-collapse* 556
*default-profiler-cutoff* 556
*default-profiler-limit* 557
*default-profiler-sort* 557
*default-simple-process-priority* 805
*default-stack-group-list-length* 117, 1090
*default-update-objects-maxlen* 930
*defstruct-generates-print-object-method* 140
*defsystem-verbose* 694
*describe-length* 27, 696
*describe-level* 696
Index

*describe-print-length* 26, 698
*describe-print-level* 26, 698
*directory-link-transparency* 1097
*disable-trace* 560
dspec-classes* 501
*enter-debugger-directly* 703
*extended-spaces* 777, 1100
*external-formats* 711
*features* 416
*file-encoding-detection-algorithm* 298, 1101
*file-eol-style-detection-algorithm* 298, 1102
*filename-pattern-encoding-matches* 1103
gesture-spec-accelerator-bit 1107
gesture-spec-control-bit 1107
gesture-spec-hyper-bit 1108
gesture-spec-meta-bit 1109
gesture-spec-shift-bit 1111
gesture-spec-super-bit 1111
grep-command* 718
grep-command-format* 719
grep-fixed-args* 720
*handle-existing-action-in-action-list* 80, 720
*handle-existing-action-list* 80, 721
*handle-existing-depackage* 582
*handle-missing-action-in-action-list* 81, 722
*handle-missing-action-list* 81, 721
*handle-old-in-package* 584
*handle-old-in-package-used-as-make-package* 584
*handle-warn-on-redefinition* 138, 722
*hidden-packages* 23, 470
*init-file-name* 724
*initialized-database-types* 224, 948
*initial-processes* 147, 162, 168, 811
*inspect-print-length* 27
*inspect-print-level* 27
*inspect-through_gui* 724
*latin-1-code-pages* 1204
*line-arguments-list* 302, 1142
*lispworks-directory* 726
*load-fasl-or-lisp-file* 585
*main-process* 824
*maximum-ordinary-windows* 137
*max-trace-indent* 45, 588
*mt-random-state* 733
*mysql-library-directories* 228, 229, 961
*mysql-library-path* 228, 229, 962
*packages-for-warn-on-redefinition* 138, 591
*print-binding-frames* 23, 475
*print-catch-frames* 23, 477
*print-command* 737
*print-handler-frames* 23, 478
*print-nickname* 737
*print-non-symbol-frames* 24
*print-open-frames* 479
*print-restart-frames* 24, 480
*print-symbols-using-bars* 1164
*process-initial-bindings* 163, 841
*profiler-print-out-all* 595
*profiler-threshold* 597
*profile-symbol-list* 123, 598
*prompt* 8, 738
*record-source-files* 516
*redefinition-action* 138, 516
*require-verbose* 744
*sg-default-size* 1191
*source-found-action* 136
*sql-enlarge-static* 1017
*sql-libraries* 224, 1018
*sql-loading-verbose* 224, 234, 1019
*stack-overflow-behaviour* 1193
*step-compiled* 446
*step-filter* 447
*step-print-env* 447
*symbol-alloc-gen-num* 111, 116, 638
*terminal-debugger-block-multiprocessing* 481
*traced-arglist* 37, 38, 40, 46, 641
*traced-results* 38, 46, 452, 642
*trace-indent-width* 46, 643
*trace-level* 46, 644
*trace-output* 46
*trace-print-circle* 46, 645
*trace-print-length* 26, 46, 646
*trace-print-level* 26, 46, 647
*trace-print-pretty* 47, 468
*trace-verbose* 649
vector-pop function 172, 174
vector-push function 172, 174
vector-push-extend function 172, 174
verbose backtrace 14
virtual (ordinary) slots 239
Index

virtual time 123

W
wait-for-input-streams function 1197
wait-for-input-streams-returning-first function 1199
wait-processing-events function 891
wait-serial-port-state function 907
weak
arrays 617
hash tables 425
weak hash tables 425
weak pointers 617
web browser 1160
:when keyword 42
when-let macro 775
when-let* macro 776
whitespace-char-p function 777
who-binds function 655
who-calls function 656
who-references function 657
who-sets function 657
Windows code page 936 296
Windows registry
API 314, 1211
Windows XP themes 1206
windows-cp936 296
with-action-item-error-handling
macro 777
with-action-list-mapping macro 779
with-dde-conversation macro 211, 1254
with-debugger-wrapper macro 484
with-exclusive-lock macro 180, 892
with-hash-table-locked 658
with-hash-table-locked macro 172, 173
with-heavy-allocation macro 112, 659
with-interrupts-blocked macro 164,
893
with-lock macro 180, 894
with-modification-change macro 174,
1199
with-modification-check-macro
macro 174, 1200
with-noticed-socket-stream macro 385
with-other-threads-disabled
macro 165, 1201
without-interrupts macro 165, 896
without-preemption function 165
without-preemption macro 165, 897
with-output-to-fasl-file function 659
with-output-to-string macro 461
with-registry-key macro 314, 1225

Y
yellow pages 350
yield function 897