Contents

Built-in Predicates
Predicate Index
Predicate Categories

Additional Information
Command-Line Switches
Errors and Error Numbers
Window Style Names
Window Message Names
Window Message Integer Values
Technical Support

Predicate Categories

Arithmetic Functions
Program Control
File Handling
Data Handling
Input and Output
Dictionaries
DOS Handling
Error Handling
Debugging
Definite Clause Grammar
Optimising Programs
Garbage Collection and Memory
Configuring Prolog
Programmable Hooks and Handlers
Windows Handling

Predicate Index - Symbolic Predicates

./2
!/0
->/2
:/2
</2
<~/2
=../2
=/2
=:=/2
=</2
==/2
=\=/2
>/2
>=/2
@</2
@=</2
@>/2
@>=/2
\+/1
\=/2
\==/2
^/2
~>/2

Predicate Index - A

abolish/1
abolish/2
abolish_files/1
'?ABORT?'/0
abort/0
abort_hook/0
absolute_file_name/2
absolute_file_name/3
abtbox/3
ansoem/2
append/3
arg/3
assert/1
assert/2
asserta/1
assertz/1
at_end_of_file/0
at_end_of_line/0
atom/1
atom_chars/2
atom_string/2
atomic/1
attrib/2

Predicate Index - B

bagof/3
beep/2
'?BREAK?'/1
break/0
break_hook/1

Predicate Index - C

'C'/3
call/1
call/2
call_dialog/2
callable/1
cat/3
catch/2
catch/3
'?CHANGE?'/3
change_hook/3
char/1
chars/1
chdir/1
chgbox/3
clause/2
clause/3
clauses/2
close/1
cmp/3
compare/3
compile/1
compound/1
consult/1
copy/2
copy_term/2
current_atom/1
current_op/3
current_predicate/1
current_predicate/2

Predicate Index - D

date/3
date/4
dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_dll_file_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0
'?DEBUG?'/1
debug/0
debug_hook/1
debugging/0
def/3
defs/2
del/1
dict/1
dir/3
dirbox/4
display/1
'?DLL?'/3
dll_hook/3
dos/0
dos/1
drive/1
dynamic/1
dynamic_call/1

Predicate Index - E

elex/1
ensure_loaded/1
env/2
eprint/1
eprint/2
eprint/3
eqv/2
erase_status_box/0
eread/1
eread/2
'?ERROR?'/2
error_hook/2
error_message/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
exec/3
expand_dcg/2
expand_term/2

Predicate Index - F

fail/0
false/0
fclose/1
fcreate/3
fdict/1
file_search_path/2
fileerrors/0
find/1
'?FIND?'/3
find_hook/3
findall/3
float/1
fluff/3
flush/0
fname/4
fndbox/2
fonts/1
fopen/3
forall/2
force/1
fread/4
free/9
functor/3
fwrite/4

Predicate Index - G

garbage_collect/0
garbage_collect/1
gc/0
get/1
get0/1
getb/1
getx/2
grab/1
ground/1

Predicate Index - H

halt/0
halt/1
help/3

Predicate Index - I

index/2
initialization/1
inpos/1
input/1
integer/1
integer_bound/3
is/2

Predicate Index - J

No predicates start with the letter J

Predicate Index - K

key_hook/3
keys/1
keysort/2

Predicate Index - L

lcall/4
lclose/1
ldict/1
leash/2
leashed/2
len/2
length/2
library_directory/1
listing/0
listing/1
load_files/1
load_files/2
lopen/1
lwrupr/2

Predicate Index - M

mem/3
member/2
member/3
message_box/3
'?MESSAGE?'/4
message_hook/4
mkdir/1
ms/2
msgbox/4
multifile/1

Predicate Index - N

name/2
nl/0
no_style_check/0
no_style_check/1
nodebug/0
nofileerrors/0
nogc/0
nonvar/1
nospy/1
nospyall/0
not/1
notrace/0
number/1
number_atom/2
number_chars/2
number_string/2
numbervars/3

Predicate Index - O

occurs_chk/2
one/1
op/3
open/2
optimize/1
optimize_files/1
otherwise/0
outpos/1
output/1

Predicate Index - P

pdict/3
phrase/2
phrase/3
portray_clause/1
predicate_property/2
print/1
printq/1
profile/4
prolog_flag/2
prolog_flag/3
prolog_load_context/2
prompt/2
prompts/2
put/1
putb/1
putx/2

Predicate Index - Q

No predicates start with the letter Q

Predicate Index - R

read/1
reconsult/1
remove/3
removeall/3
ren/2
repeat/0
repeat/1
retract/1
retract/2
retractall/1
reverse/2
rmdir/1

Predicate Index - S

save_predicates/2
see/1
seed/1
seeing/1
seen/0
setof/3
show_dialog/1
simple/1
skip/1
skip_layout/0
skip_line/0
skip_term/0
solution/2
sort/2
sort/3
source_file/1
source_file/2
source_file/3
spy/1
stamp/2
statistics/0
statistics/2
stats/4
status_box/1
stream_position/2
stream_position/3
string/1
string_chars/2
sttbox/2
stuff/3
style_check/0
style_check/1
subsumes_chk/2
switch/2
sysops/0
system_menu/3

Predicate Index - T

tab/1
tell/1
telling/1
term_expansion/2
throw/2
ticks/1
time/4
timer/2
'?TIMER?'/3
timer_hook/3
told/0
total/9
trace/0
true/0
ttyflush/0
ttyget/1
ttyget0/1
ttynl/0
ttyput/1
ttyskip/1
ttytab/1
type/2

Predicate Index - U

unifiable/2
unknown_predicate_handler/2

Predicate Index - V

var/1
vars/2
ver/1
ver/4
volatile/1

Predicate Index - W

wait/1
warea/5
wbclose/1
wbdict/1
wbload/2
wbopen/2
wbtnsel/2
wbusy/1
wcclose/1
wccreate/8
wcdict/1
wclass/2
wclose/1
wcopen/2
wcount/4
wcreate/8
wdcreate/7
wdict/1
wedtclp/2
wedtfnd/6
wedtlin/4
wedtpxy/4
wedtsel/3
wedttxt/2
wenable/2
wfclose/1
wfcreate/4
wfdata/5
wfdict/1
wfind/3
wflag/1
wfocus/1
wfont/2
wfsize/4
wgfx/2
wgfx/6
wgfxadd/5
wgfxcur/2
wgfxget/5
wgfxmap/5
wgfxorg/3
wgfxpnt/1
wgfxsub/5
wgfxtst/5
wiclose/1
widict/1
wiload/3
winapi/5
window_handler/2
window_handler/4
wiopen/2
wlbxadd/3

wlbxdel/2
wlbxfnd/4
wlbxget/3
wlbxsel/3
wlink/3
wmclose/1
wmcreate/1
wmdict/1
wmnuadd/4
wmnudel/2
wmnuget/4
wmnunbl/3
wmnusel/3
wndhdl/2
wprnend/1
wprngfx/1
wprngfx/5
wprnini/4
wprnmap/4
wprnorg/2
wprnpag/1
wprnres/4
wprnstt/1
wrange/4
write/1
write_canonical/1
writeq/1
wshow/2
wsize/5
wstyle/2
wtcreate/6
wtext/2
wthumb/3
wucreate/6
wxclose/1
wxcreate/6
wxdict/1
wxload/2
wxsave/2

Predicate Index - X

No predicates start with the letter X

Predicate Index - Y

No predicates start with the letter Y

Predicate Index - Z

No predicates start with the letter Z

Arithmetic

</2	*expression less than*
=:=/2	*expression equality*
=</2	*expression less than or equal*
=\=/2	*expression inequality*
>/2	*expression greater than*
>=/2	*expression greater than or equal*
is/2	*expression evaluator*
seed/1	*re-seed the random number generator*

Program Control

Control

!/0	*control backtracking*
,/2	*conjunction*
->/2	*if then*
;/2	*disjunction*
\+/1	*negation*
abort/0	*abort the current program*
fail/0	*force failure*
false/0	*force failure*
halt/0	*terminate the current Prolog session.*
halt/1	*terminate the current Prolog session and return an error code.*
not/1	*logical negation.*
otherwise/0	*succeed.*
repeat/0	*succeed even on backtracking.*
repeat/1	*succeed even on backtracking for a given number of times*
true/0	*succeed*
break/0	*suspend the current execution*
'?BREAK?'/1	*user-defined Prolog program which intercepts break messages*
break_hook/1	*built-in break hook*

Looking at the Program State

current_atom/1	*check or get a current atom*
current_predicate/1	*check or get a current predicate*
current_predicate/2	*check or get a current predicate*
current_op/3	*get the name, type and precedence of a currently defined*

	operator
def/3	*test for a currently defined predicate and return its type*
defs/2	*return all arities for predicate*
pdict/3	*return a dictionary of predicates*
predicate_property/2	*find the association between predicates and properties*

Meta-Programming

=../2	*defines the relationship between a structure/atom and a list*
arg/3	*find the nth argument of a term*
call/1	*call a prolog goal.*
call/2	*call a Prolog goal and return the termination port*
functor/3	*the relationship between a term its functor name and its arity*
one/1	*one solution meta-call.*

Sets of Solutions

^/2	*existential quantifier*
bagof/3	*find all the instances of a term for which a prolog goal is true*
findall/3	*find all the instances of a term for which a prolog goal is true*
forall/2	*generate then test solutions for a goal*
setof/3	*find the set of instances of a term for which a prolog goal is true*
solution/2	*return the nth soution to a specified call*

Program Timing

date/3	*system date*
date/4	*date to day number conversion*
ms/2	*time a given Prolog goal*
ticks/1	*get a time reference*
time/4	*get the system time*

File Handling

Files and Directories

absolute_file_name/2	*converts from a relative to an absolute file specification*
absolute_file_name/3	*convert between a relative and an absolute file specification using options*
cat/3	*atom and string concatenation*

close/1	*close the named file*
fclose/1	*close a file*
fdict/1	*return a dictionary of files*
file_search_path/2	*user-defined fact specifying a path name*
fname/4	*convert a file name into parts*
fopen/3	*open a file with the given access mode*
library_directory/1	*defines a library directory*
mkdir/1	*make a directory*
open/2	*open a file with the given access mode*
attrib/2	*set or get file attributes*
chdir/1	*choose or return a directory*
del/1	*delete a file*
dir/3	*get a file directory*
drive/1	*choose or return a drive*
env/2	*get an environment string*
fcreate/3	*create a file with the given attributes*
ren/2	*rename a file*
rmdir/1	*delete a directory*
stamp/1	*get or set a file date and time stamp*

Loading and Saving

abolish_files/1	*abolish all predicates associated with the given file*
compile/1	*compile the specified Prolog source file(s) into object code format*
consult/1	*load a source code program into memory*
ensure_loaded/1	*load the specified Prolog source and/or object file(s) into memory.*
initialization/1	*declare a goal to be run on loading a file*
load_files/1	*load the specified Prolog source and/or object files*
load_files/2	*load the specified Prolog source and/or object files using certain options.*
multifile/1	*allow the specified predicates to be defined in more than one file*
prolog_load_context/2	*find the context of the current load*
reconsult/1	*load a source code program into memory replacing the previous version*
save_predicates/2	*save the specified predicates to a file in object code format.*
source_file/1	*check or get the files that are currently loaded*

source_file/2	*check or get the predicates associated with currently loaded files*
source_file/3	*as source_file/2 but returns the numbers of the clauses also*

Data Handling

List Handling

append/3	*join or split lists*
length/2	*get the length of a Prolog list*
mem/3	*return the given member of a term*
member/2	*get or check a member of a list*
member/3	*get or check a member of a list and its position in the list*
remove/3	*remove an element from a list*
removeall/3	*remove all occurrences of an item from a list*
reverse/2	*check or get the reverse of a list*

String Handling

<~/2	*re-direct input to a file or a string*
~>/2	*re-direct output to a file or a string*
cat/3	*atom and string concatenation*
elex/1	*set, reset or get the edinburgh syntax flag*

Term Comparison and Sorting

=/2	*unification between two terms*
==/2	*check that two terms are identical*
@</2	*check that one term is less than another*
@=</2	*check that one term is equal to or less than another*
@>/2	*check that one term is greater than another*
@>=/2	*check that one term is greater than or equal to another*
\=/2	*tests for non-unification between two terms*
\==/2	*check that two terms are not identical*
cmp/3	*compare two terms*
compare/3	*find the relationship between one term and another*
eqv/2	*check two terms for equivalence*
keysort/2	*sort a list of key-value pairs into ascending order*
len/2	*return length of a term*
sort/2	*sort a list into ascending order and remove duplicates*
sort/3	*sort a list into ascending order using a key, do not remove*

	duplicates
occurs_chk/2	*occurs check*
subsumes_chk/2	*check that one term subsumes another*

Term Conversion

=../2	*defines the relationship between a structure/atom and a list*
atom_chars/2	*converts between an atom and a list of characters*
atom_string/2	*convert from an atom to an LPA string*
copy_term/2	*copy a term with new variables*
lwrupr/2	*convert between lower and upper case*
name/2	*convert between an atom or number and a byte list*
number_atom/2	*convert between a number and an atom*
number_chars/2	*convert between a number and a list of characters*
number_string/2	*convert between a number and an LPA string*
numbervars/3	*instantiate the variables in a given term*
string_chars/2	*convert from a list of ASCII character codes to an LPA string*

Term Input and Output

current_op/3	*get the name, type and precedence of a currently defined operator*
display/1	*write a term to the standard output stream in standard prefix notation*
elex/1	*set, reset or get the edinburgh flag*
eprint/1	*print a quoted edinburgh term to the current output stream*
eprint/2	*same as eprint/1 but with the ability to output variable names*
eprint/3	*same as eprint/2 but with added priority*
eread/1	*read an edinburgh term from the current input stream*
eread/2	*same as eread/1 but with an added variable list*
etoks/1	*read an edinbugh token list from the current input stream*
etoks/2	*read an edinbugh token list from the current input stream with variable names*
ewrite/1	*write an unquoted edinburgh term to the current output stream*
ewrite/2	*same as ewrite/1 but with the ability to output variable names*
ewrite/3	*same as ewrite/2 but with added priority*
op/3	*declare an operator with a given precedence and type*

portray_clause/1	*write a clause to the current output stream in listing format*
print/1	*print a term to the current output stream*
printq/1	*print a quoted term to the current output stream*
read/1	*read a term from the current input stream*
sysops/0	*re-install all of the system-declared operators*
skip_term/0	*skip the remaining input characters up to the end of a term*
vars/2	*return a named list of vars in a term*
write/1	*write a term to the current output stream*
write_canonical/1	*write a term to the current output stream in canonical form*
writeq/1	*write a quoted term to the current output stream*
prompt/2	*get or set the Prolog prompt*

Term Type Checking

atom/1	*test for an atom*
atomic/1	*test for an atom or a number*
callable/1	*check to see if a term is an atom or a compound.*
char/1	*check for an integer in the range 0 - 255*
chars/1	*check for a list of integers in the range 0 - 255*
compound/1	*test for a compound term*
float/1	*test for a floating point number*
ground/1	*check for completely bound terms*
integer/1	*test for an integer*
integer_bound/3	*generate or test a number between lower and upper bounds*
nonvar/1	*test for a non-variable*
number/1	*test for a floating point number or integer*
simple/1	*check for an atom, number or variable*
string/1	*test for a string*
type/2	*return type of a term*
unifiable/2	*check that two terms are potentially unifiable*
var/1	*test for an uninstantiated variable*

The Clause Database

abolish/1	*delete all the predicates specified by the given argument*
abolish/2	*delete all clauses for the given predicate and arity*
abolish_files/1	*abolish all predicates associated with the given file*
assert/1	*add a clause at the end of the clauses associated with its predicate name*

assert/2	*assert the clause at the given position*
asserta/1	*add a clause at the beginning of the clauses associated with its predicate name*
assertz/1	*add a clause at the end of the clauses associated with its predicate name*
clause/2	*get or check the body of a clause given its head*
clause/3	*find the position of a clause in a dynamic predicate*
clauses/2	*return a list of candidates for a dynamic predicate that match a head*
dynamic/1	*define a predicate to be dynamic*
dynamic_call/1	*call a dynamic procedure safely*
functor/3	*the relationship between a term its functor name and its arity*
listing/0	*list all the dynamic clauses in the workspace to the current output stream*
listing/1	*list the specified dynamic predicates to the current output stream*
retract/1	*delete a clause that matches the given clause*
retract/2	*retract a clause at a specified position*
retractall/1	*delete all clauses that match the given clause head*
volatile/1	*declare that the clauses for a predicate will not be saved in object files*

Data Compression and Decompression

stuff/3	*compress the data in the current input stream to the current output stream*
fluff/3	*decompress the data in the current input stream to the current output stream*

Input and Output

Predicates for Setting I/O Streams

input/1	*set input from a file, device or string*
output/1	*set output to a file, device or string*
see/1	*set the current input stream*
seeing/1	*return the current input stream*
seen/0	*reset the current input stream to the standard input stream*
tell/1	*set the current output stream*
telling/1	*return the current output stream*
told/0	*reset the current output stream to the standard output*

	stream

Predicates for Temporarily Redirecting I/O

<~/2	*re-direct input to a file or a string*
~>/2	*re-direct output to a file or a string*

Predicates for Positioning File Pointers

at_end_of_file/0	*checks to see if the input file pointer is at end of file*
at_end_of_line/0	*test whether end of line has been reached for the current input stream.*
find/1	*find a string or atom in a file*
inpos/1	*set the input stream position*
outpos/1	*sets the output stream position*
skip/1	*skip to just after the specified ASCII value on the current input stream*
skip_layout/0	*skip past the white space characters on the current input stream*
skip_line/0	*skip the remaining input characters of the current line*
skip_term/0	*skip the remaining input characters up to the end of a term*
stream_position/2	*get the current position of the specified stream*
stream_position/3	*get the current position of the specified stream*
flush/0	*flush the current input stream*

Formatted I/O Predicates

fread/4	*formatted read of a term*
fwrite/4	*formatted write of a term*

Character I/O Predicates

get/1	*read a non-white-space character from the current input stream*
get0/1	*read a character from the current input stream*
getx/2	*input a byte, word or dword*
put/1	*write an ASCII character to the current output stream*
putx/2	*output a byte, word or dword to the current output stream*
getb/1	*get a byte direct from keyboard*
putb/1	*byte output direct to screen*

Predicates for Outputting Format Characters

nl/0	*start a new line on the current output stream*
tab/1	*write the given number of spaces to the current output*

	stream

Predicate for Copying Data From File To File

copy/2	*copy data from the current input stream to the current output stream*

Keyboard and Screen I/O

keys/1	*get the system key status*
grab/1	*check for a byte direct from keyboard*
ttyflush/0	*flush the user output stream*
ttyget/1	*read a non-white-space character from the user input stream*
ttyget0/1	*read a character from the user input stream*
ttynl/0	*start a new line on the user ouput stream*
ttyput/1	*write an ASCII character to the user output stream*
ttyskip/1	*skip to just after the specified ASCII value on the user input stream*
ttytab/1	*write the given number of spaces to the user output stream*

Sound Output Predicates

beep/2	*sound a beep of the given duration and frequency*

Dictionaries

dict/1	*return the current atoms*
fdict/1	*return the open files*
pdict/3	*return the current predicates*
wfdict/1	*return the open fonts*
ldict/1	*return the open dynamic link libraries*
wmdict/1	*return the defined menus*
wdict/1	*return the currently open windows*
wbdict/1	*return the open bitmaps*
wcdict/1	*return the open cursors*
widict/1	*return the current icons*
wxdict/1	*return the current Windows meta files*

DOS Handling

dos/0	*initiate a DOS shell*
dos/1	*initiate a DOS shell and run the given command*
exec/3	*execute an external program*
switch/2	*set or get the value of a LPA command line switch*

ver/4	*return information on the current version of Prolog*

Error Handling

abort/0	*abort the current program*
error_message/2	*return an error message for an error number*
unknown_predicate_handler/2	*user-defined fact that defines the handling of unknown predicates*
catch/2	*catch the error code generated by a given goal*
catch/3	*same as catch/2 but also return the predicate that generated the error*
throw/2	*throw a numbered error*
'?ERROR?'/2	*user-defined error handler*
error_hook/2	*system defined behaviour for error handling*
flush/0	*flush the current input stream*

Debugging

debug/0	*set the debug mode to on*
'?DEBUG?'/1	*user-defined Prolog program which intercepts calls to the debugger*
debug_hook/1	*system handler for the debug hook*
debugging/0	*write the current status of the debugger to the standard output stream*
force/1	*call a Prolog goal and suspend the debugger for that call*
leash/2	*set the interaction with the debugger*
leashed/2	*test or get the leashes on the debugging ports*
ms/2	*time a given Prolog goal*
no_style_check/0	*turn off all compile-time style checking*
no_style_check/1	*turn off the specified style of compile-time style checking*
nodebug/0	*switch the debug mode to off*
nospy/1	*remove the spy points from the specified predicates*
nospyall/0	*remove all spy points*
notrace/0	*turn the debug mode to off*
spy/1	*set a spy point on the specified predicates*
style_check/0	*turn on all compile-time style checking.*
style_check/1	*turn on the specified type of compile-time style checking.*
trace/0	*switch the trace mode to on*

Definite Clause Grammar

'C'/3	*used in the expansion of grammar rules*

expand_dcg/2	*convert grammar rules to Prolog without calling term_expansion/2*
expand_term/2	*convert between a grammar rule and its Prolog equivalent*
phrase/2	*checks if a sequence of symbols can be parsed as a given type*
phrase/3	*checks if a sequence of symbols can be parsed as a given type*
term_expansion/2	*user-defined hook for grammar rule translation*

Optimising Programs

index/2	*declare multiple argument indexes*
optimize/1	*optimize a static predicate*
optimize_files/1	*file to file optimization of code*

Garbage Collection and Memory

free/9	*return the free space available in Prolog's memory areas*
total/9	*return the total space allocated to Prologs memory areas*
garbage_collect/0	*invokes the garbage collector explicitly*
garbage_collect/1	*invoke the garbage collection of the given memory area*
gc/0	*enable the garbage collector*
nogc/0	*disable the garbage collector*
statistics/0	*display statistics about the current status of the system*
statistics/2	*get individual memory statistics*
stats/4	*get assorted runtime statistics*
ver/4	*return information on the current version of Prolog*
ver/1	*output the standard banner*

Configuring Prolog

fileerrors/0	*turn on the reporting of file error messages*
no_style_check/0	*turn off all the compile-time style checking*
no_style_check/1	*turn off the specified style of compile-time style checking*
nofileerrors/0	*turn off the reporting of file error messages*
prolog_flag/2	*get or check the values for global environment variables*
prolog_flag/3	*set and get values for global environment variables*
style_check/0	*turn on all the compile-time style checking*
style_check/1	*turn on the specified type of compile-time style checking*
prompt/2	*get or set the Prolog prompt*
prompts/2	*get or set the buffered console input prompts*
switch/2	*set or get the value of an LPA Prolog command line switch*

Pre-defined Dialogs

message_box/3	*create a message box and return a response*
status_box/1	*display a status message window*
erase_status_box/0	*destroy the status message window*

Programmable Hooks and Handlers

Built-in Hooks

'?ABORT?'/0	*user-defined Prolog program which intercepts program aborts*
abort_hook/0	*system hook for handling program aborts*
'?BREAK?'/1	*user-defined Prolog program which intercepts break messages*
break_hook/1	*system hook for handling program breaks*
'?CHANGE?'/3	*user-defined hook for handling change box messages*
change_hook/3	*system hook for handling change box messages*
'?DEBUG?'/1	*user-defined Prolog program which intercepts calls to the debugger*
debug_hook/1	*system hook for the debugger*
'?DLL?'/3	*user-defined hook for handling DLL messages*
dll_hook/3	*system hook for handling DLL messages*
'?ERROR?'/2	*user-defined Prolog program which intercepts error messages*
error_hook/2	*system hook for handling error messages*
'?FIND?'/3	*user-defined hook for handling find box messages*
find_hook/3	*system hook for handling find box messages*
'?MESSAGE?'/4	*user-defined Prolog program which intercepts messages*
message_hook/4	*system hook for handling window messages*
'?TIMER?'/3	*user-defined hook for the 64 built-in timers*
timer_hook/3	*system hook for handling timer interrupts*

Windows Handling

Bitmap Handling
Built-In Dialogs
Built-in Hooks
Button Handling
Cursor Handling
DDE Support
Dialog Handling
DLL Handling
Edit Control Handling
Error Handling
Event Handling
Fonts and Character Sets
Graphics Handling
Icon Handling
Listbox Handling
Menu Handling
Printer Handling
Profile Handling
Prolog Environment Handling
Scrollbar Handling
Text Handling

Text Window Handling
User Window Handling
Window Handling
Windows API Functions
Windows Help Function
Windows Metafile Handling

Built-In Dialogs

abtbox/3	*display the about box*
chgbox/3	*display the modeless change box*
'?CHANGE?'/3	*user-defined hook for handling change box messages*
change_hook/3	*system handler for the change dialog*
dirbox/4	*display the directory box*
erase_status_box/0	*destroy the status message window*
'?FIND?'/3	*user-defined hook for handling find box messages*
find_hook/3	*system hook for handling find box messages*
fndbox/2	*display the modeless find box*
message_box/3	*create a message box and retrun a response*
msgbox/4	*display the message box*
status_box/1	*display a status message window*
sttbox/2	*display or update a status box*

DDE Support

dde_advise_dict/1	*get or check the list of open advise loops*
dde_channel_dict/1	*get or check a list of open source channels*
dde_close/1	*close a source channel*
dde_close_advise/2	*close an advise loop for a given open channel*
dde_close_all/0	*close all source channels*
dde_close_all_topics/0	*close all registered topics*
dde_close_topic/1	*close a named topic*
dde_dll_name/1	*gets or checks the absolute file name of the DDE Dynamic Link Library*
dde_dll_file_name/1	*user-defined fact for setting the absolute file name of the DDE Dynamic Link Library*
dde_enable_state/2	*Get or set the enable state of a channel*
dde_execute/2	*start an execute transaction*
dde_fetch_data/1	*fetch data for a transaction*
dde_load/0	*load the DDE Dynamic Link Library*
dde_open/3	*open a DDE source channel*
dde_open_advise/4	*open an advise loop*
dde_open_topic/2	*open a topic*
dde_poke/3	*poke data to a channel*
dde_put_data/1	*put data to a channel*
dde_request/3	*a DDE request transaction.*
dde_timeout/1	*get or set the DDE time out value*
dde_topic_dict/1	*get or check a list of open topics*
dde_unload/0	*unload the DDE Dynamic Link Library*

Fonts and Character Sets

ansoem/2	*convert between ansi and oem strings*
fonts/1	*return a list of available fonts*
wfclose/1	*close a font*
wfcreate/4	*create a font*
wfdata/5	*check or get the typeface, size, style and ascent of the given logical font*
wfdict/1	*return a dictionary of fonts*
wfont/2	*get or set the font of a window*
wfsize/4	*check or get the height and width of the given string in the given font*

Built-in Hooks

abort_hook/0	*system hook for handling program aborts*
break_hook/1	*system hook for handling program breaks*
change_hook/3	*system hook for handling change box messages*
debug_hook/1	*system hook for the debugger*
dll_hook/3	*system hook for handling DLL messages*
error_hook/2	*system hook for handling error messages*
find_hook/3	*system hook for handling find box messages*
message_hook/4	*system hook for handling window messages*
timer_hook/3	*system hook for handling timer interrupts*

Dialog Handling

call_dialog/2	*call a modal dialog and get or check the result*
show_dialog/1	*run a window as a modeless dialog*
wccreate/8	*create a control window*
wdcreate/7	*create a dialog window*
window_handler/2	*get or set the current message handler for the given window*
window_handler/4	*system defined handler for windows*

DLL Handling

'?DLL?'/3	*user-defined hook for handling DLL messages*
dll_hook/3	*system hook for handling DLL messages*
lcall/4	*call a dynamic link library function*
lclose/1	*close a dynamic link library*
ldict/1	*return a list of all currently open dynamic link libraries*
lopen/1	*open a dynamic link library*
winapi/5	*call a C function defined in the Windows API environment or in a DLL*

Menu Handling

wmclose/1	*close a menu*
wmcreate/1	*create a menu*
wmdict/1	*return a dictionary of menus*
wmnuadd/4	*add an item to a menu*
wmnudel/2	*delete an item from a menu*
wmnuget/4	*get an item from a menu*
wmnunbl/3	*get or set enable status of an item on a menu*
wmnusel/3	*get or set selection state of an item on a menu*

Profile Handling

profile/4	*get or set a profile string*

Event Handling

timer/2	*get or set the status of the given timer interrupt*
'?TIMER?'/3	*user-defined hook for the 64 built-in timers*
timer_hook/3	*system hook for handling timer interrupts*
wait/1	*get or set the window message status*
wbusy/1	*get or set the busy cursor flag*
wflag/1	*get or set the Windows message interrupt flag*

Window Handling

warea/5	*get or check client area size and position*
wclass/2	*check or get the class of a given window*
wclose/1	*close a window*
wccreate/8	*create a control window*

wcreate/8	*create a window*
wdcreate/7	*create a dialog window*
wdict/1	*get all currently open windows*
wenable/2	*get or set window enable status*
wfind/3	*find the handle for a named window*
wfocus/1	*get or set input focus to a window*
wlink/3	*find the handle for a linked window*
wndhdl/2	*convert between window and handle*
wshow/2	*get or set show or hide status*
wsize/5	*get or set window size and position*
wstyle/2	*get or set window style*
wtext/2	*get or set the window text*
wtcreate/6	*create a text window*
wucreate/6	*create a user MDI window*

Windows API Functions

winapi/5	*call a Windows API environment C function*

Bitmap Handling

wbclose/1	*close a bitmap*
wbdict/1	*return a dictionary of bitmaps*
wbload/2	*load a bitmap from a disk file*
wbopen/2	*load a bitmap from local resources*

Button Handling

wbtnsel/2	*get or set selection state of a button*

Cursor Handling

wcclose/1	*close a cursor*
wccreate/8	*create a control window*
wcdict/1	*return a dictionary of cursors*
wcopen/2	*load a cursor from local resources*

Text Handling

wcount/4	*get char, word and line counts for the given window*
wtext/2	*get or set the window text*

Edit Control Handling

wedtclp/2	*perform a clipboard function*
wedtfnd/6	*find a text string in an "edit" or "editor" control window*
wedtlin/4	*get offsets a line in an "edit" or "editor" control window*
wedtpxy/4	*convert between linear offset and x, y coordinates in "edit" or "editor" windows*
wedtsel/3	*get or set selection in an "edit" or "editor" control window*
wedttxt/2	*get or set the text of the given "edit" or "editor" window*

Graphics Handling

wgfx/2	*perform a windows graphics sequence*
wgfx/6	*perform a clipped windows graphics sequence*
wgfxadd/5	*add rectangle to graphics update region*
wgfxcur/2	*get or set the cursor for a grafix window*
wgfxget/5	*get the graphics update region*
wgfxmap/5	*get or set the graphics mapping*
wgfxorg/3	*get or set the graphics origin for a given window*

wgfxpnt/1	*force the painting of the graphics update region*
wgfxsub/5	*subtract a rectangle from a graphics update region*
wgfxtst/5	*perform a windows graphics hit test*

Icon Handling

wiclose/1	*close an icon*
widict/1	*return a dictionary of icons*
wiload/3	*load an icon from a disk file*
wiopen/2	*load an icon from local resources*

Listbox Handling

wlbxadd/3	*add an item to a list box*
wlbxdel/2	*delete an item from a list box*
wlbxfnd/4	*find a string in a list box*
wlbxget/3	*get an item from a list box*
wlbxsel/3	*get or set selection in a list box*

Scrollbar Handling

wrange/4	*get or set range of a scroll bar*
wthumb/3	*get or set position of a scroll bar*

Windows Metafile Handling

wxclose/1	*close a metafile*
wxcreate/6	*create a metafile*
wxdict/1	*return a dictionary of metafiles*
wxload/2	*load a metafile from disk*
wxsave/2	*save a metafile to disk*

Text Window Handling

wtcreate/6	*create a text window*

User Window Handling

wucreate/6	*create a user MDI window*

Printer Handling

wprnend/1	*finish or abort the use of the printer*
wprngfx/1	*perform a printer graphics sequence*
wprngfx/5	*perform a clipped printer graphics sequence*
wprnini/4	*initialise the printer*
wprnmap/4	*get or set the printer graphics mapping*
wprnorg/2	*get or set the printer graphics origin*
wprnpag/1	*start a new printer page*
wprnres/4	*get or check the printer resolution*
wprnstt/1	*get or check the printer status*

Prolog Environment Handling

system_menu/3	*invoke a Prolog environment menu function*

Windows Help Function

help/3	*perform a Windows help function*

Argument types

<arity>
<atom>
<char>
<char_list>
<clause>
<compound_term>
<conjunct_of Type>
<expr>
<file_name>
<file_spec>
<file_specs>
<float>
<functor>
<goal>
<integer>
<integer_expr>
<list>
<list_of Type>
<number>
<path_alias>
<pred_spec>
<pred_specs>
<string>
<term>
<variable>
<window_handle>

abtbox/3 styles

Value	Style
0	fixed IBM PC font with large Prolog bitmap
1	proportional Windows font with large Prolog bitmap
2	fixed IBM PC font without bitmap
3	proportional Windows font without bitmap

Styles 4-7 have the same attributes as above except that the window displayed is wider. In this format a smaller LPA bitmap is displayed on the left hand side of the window and the text is displayed on the right.

chgbox/3 flags

Value	Action
-1	Destroy the change box
0	Hide the change box
1	Display and enable the change box (with 'Change' and 'Change+Find' buttons disabled).
2	Display and enable the change box (with all buttons enabled).

chgbox/3 messages

Value	Action
msg_cbclose	User wants to close the box
msg_cbfind	User wants to perform a find
msg_cbfndnxt	User wants to perform a find next
msg_cbchange	User wants to perform a change
msg_cbchgfnd	User wants to perform a change+find
msg_cbchgall	User wants to perform a change all

chgbox/3 radio button values

Value	Meaning
1	Apply changes to selected area only
2	Apply changes to whole of current text window
3	Apply changes to all text windows

Term comparison relationships

Value	Relationship
=	Term1 == Term2.
<	Term1 @< Term2.
>	Term1 @> Term2.

def/3 predicate type values

Value	Type
0	null predicates
1	incrementally compiled
2	optimized
3	assembler
4	external

dir/3 file type values

Value	File Attributes
1	read/write or read only
2	hidden files
4	system files
8	volume label
16	directories
32	files with archive bit set.

Token values

Number	Type
0	variable
1	integer
2	floating point number
3	unquoted atom
4	string
5	empty list
6	list
7	quoted atom
8	punctuation character

File attribute values

Value	File Attributes
0	read/write
1	read-only
2	hidden read/write file
3	hidden read-only file

fndbox/2 flags

Value	Action
-1	Destroy the find box
0	Hide the find box
1	Display and enable the find box

fndbox/2 messages

Value	Action
msg_fbclose	User wants to close (hide) the box
msg_fbfind	User wants to perform a find operation
msg_fbfndnxt	User wants to perform a find next operation

fndbox/2 radio button values

Value	Meaning
1	Search for text in selected area only
2	Search for text in whole of current text window
3	Search for text in all text windows

File access modes

Value	Access Mode
0	read only, the quickest access mode for reading as no attempt is made to write to the file or device before reading in the next buffer full.
1	write only, the fastest access mode for writing to a file or an output device (such as a printer) because no attempt is made to read from the file or device after flushing the buffer.
2	read/write, this mode is slower than the other modes because, after flushing the buffer on output, the next buffer full must be read in. This is necessary to support the possibility of interleaved reads and writes.

formatted read and write format types

Format	Type
a	atom (uses modifier)
b	byte list (uses modifier)
f	fixed point number (uses modifier)
i	integer
n	unsigned integer
r	arbitrary radix (uses modifier)
s	string (uses modifier)

help/3 help function values

Value	Function
1	Go to the topic specified by the context number
2	Quit from the help file
3	Enter the help file at the first page
4	Open the help on help file
5	Set the index
6	Put the topic specified by the context number into a popup window
7	Force the help application to the front

Special input streams

Value	Input Stream
0	user console (buffered input)
1	buffered input (for compatibility)
2	raw terminal input (no buffering or echo)

is/2 arithmetic functions

Function	Description
X + *Y*	the sum of *X* and *Y*.
X - *Y*	the difference of *X* and *Y*.
-*X*	the negative of *X*.
X * *Y*	the product of *X* and *Y*.
X / *Y*	the quotient of *X* and *Y*.
X // *Y*	the integer quotient of *X* and *Y*. The result is truncated to the nearest integer between it and 0.
X mod *Y*	the remainder after integer division of *X* by *Y*. The result is the same sign as *X*.
X ^ *Y*	*X* to the power of *Y*.
abs(*X*)	the absolute value of *X*. e.g. abs(-3.5) returns 3.5 .
acos(*X*)	the arccosine of *X* in degrees.
aln(*X*)	e to the power of *X*.
alog(*X*)	10 to the power of *X*.
asin(*X*)	the arcsine of *X* in degrees.
atan(*X*)	the arctangent of *X* in degrees.
cos(*X*)	the cosine of *X* degrees.
fp(*X*)	the fractional part of *X*. e.g. fp(-3.5) returns -0.5 .
int(*X*)	the first integer equal to or less than *X*. e.g. int(-3.5) returns -4 .
ip(*X*)	the integer equal part of *X*. e.g. ip(-3.5) returns -3 .
ln(*X*)	the natural logarithm of *X*.
log(*X*)	the base 10 logarithm of *X*.
max(*X*,*Y*)	the maximum value of *X* and *Y*. e.g. max(3.5,4). returns 4.
min(*X*,*Y*)	the minimum value of *X* and *Y*. e.g. min(3.5,4). returns -3.5 .
rand(*X*)	a random floating point number between zero and *X*.
sign(*X*)	-1 if *X* is negative, 0 if *X* is 0, or 1 if *X* is positive. e.g. sign(-3.5) returns -1.
sin(*X*)	the sine of *X* degrees
sqrt(*X*)	the square root of *X*.
tan(*X*)	the tangent of *X* degrees.

is/2 integer bitwise arithmetic functions

Function	Description
X /\ *Y*	the logical and of the integers *X* and *Y*.
X \/ *Y*	the logical inclusive or of the integers *X* and *Y*.
X << *Y*	the logical shift arithmetic left of the integer *X* by the number *Y* bits (vacated bits are filled with zeros).
X >> *Y*	the logical shift arithmetic right of the integer *X* by the number *Y* bits (the most significant bit is propagated into the vacated bits).
\(*X*)	the logical negation of the integer *X*.
a(*X*,*Y*)	the logical and (AND) of the integers *X* and *Y*.
l(*X*,*Y*)	the logical left rotation of the integer *X* by the number *Y* bits.
o(*X*,*Y*)	the logical inclusive or (OR) of the integers *X* and *Y*.
r(*X*,*Y*)	the logical right rotation of the integer *X* by the number *Y* bits.
x(*X*,*Y*)	the logical exclusive or(XOR) of the integers *X* and *Y*.

keys/1 key values

Value	Key
1	Right <Shift> key
2	Left <Shift>
4	<Ctrl> key
8	<Alt> key
16	<Scroll Lock> toggle value

32	<Num Lock>	toggle value
64	<Caps Lock>	toggle value
128	<Ins>	toggle value
256	<Ctrl>	held down
512	<Alt>	held down
8192	<Num Lock>	held down

load_files/2 options

Option	Arg	Function
if*(Arg)*	*Arg*=true	(default) always load.
	Arg=changed	load file if it is not already loaded or if it has been changed since it was last loaded.
load_type*(Arg)*	*Arg*=source	reconsult Prolog source code.
	Arg=compile	compile Prolog source code into object code.
	Arg=object	load object code.
	Arg=latest	(default) load object files or source, whichever is newer
all_dynamic(*Arg*)	*Arg*=true	load all predicates as dynamic.
	Arg=false	(default) load predicates as static unless they are declared dynamic.

System timer resolutions

System	Timer Resolution
WIN-PROLOG	1/182 (54.92 ms)
MacProlog32	1/60 (16.67 ms)

msgbox/4 styles

The available styles are divided into four groups: buttons, icons, default button and modality. Valid styles may be made by using one member from each group and adding them together.

GROUP	STYLE	VALUE
Buttons	Ok	16'00000000
	Ok + Cancel	16'00000001
	Abort + Retry + Ignore	16'00000002
	Yes + No + Cancel	16'00000003
	Yes + No	16'00000004
	Retry + Cancel	16'00000005
Icons	None	16'00000000
	Stop Sign	16'00000010
	Question Mark	16'00000020
	Exclamation Mark	16'00000030
	Information Sign	16'00000040
Default Button	First	16'00000000
	Second	16'00000100
	Third	16'00000200
Modality	Application	16'00000000
	System	16'00001000
	Task	16'00002000
	No focus	16'00003000

32-bit stamp flag

High Word			Low Word		
bits 26-32	**bits 22-25**	**bits 17-21**	**bits 12-16**	**bits 6-11**	**bits 1-5**
Years (since 1980)	Month	Day	Hours	Minutes	Seconds

special output streams

Value	Special Stream
0	processed screen output (windows or DOS)
1	DOS processed output (insensitive to windows)
2	raw terminal output (no processing of chars)
3	mda screen output (subset of windows processing)

op/3 operator types and meanings

Operator Type	Meaning
fx	non-associative prefix operator
fy	right associative prefix operator
xf	non-associative postfix operator
yf	left associative postfix operator
xfx	non-associative infix operator
xfy	right associative infix operator
yfx	left associative infix operator

pdict/3 predicate type values

Value	Type
-1	all predicates
0	null predicates
1	incrementally compiled
2	optimized
3	assembler
4	external

Predicate property types

compiled	interpreted
built_in	multifile
foreign	volatile
spy	dynamic
static	index(Index)

prolog_flag/2 flag types and values

Flag	Default Setting	Flag	Default Setting
save_on_exit	off	skip_exit	on
context	supervisor	display_unify	bindings
print_stream	console	gc	on
print_message	modify	gc_collect	heap
print_log_file	off	advice	off
resave	when_changed	character_escapes	off
recompile	when_changed	max_depth	0
fileerrors	on	supervisor_prompt	'| ?- '
syntax_errors	dec10	consult_prompt	'| '
consult_errors	retry	user_prompt	'|: '
consult_status	off	text_extension	'.TXT'
check_single_var	off	source_extension	'.PL'
check_contiguous	off	object_extension	'.PC'
check_multiple	off	project_extension	'.PJ'
unknown	error	foreign_extension	'.O'
debugging	debug	flex_extension	'.KSL'
debug_file	srcbug	ppp_extension	'.PPP'
skip_spy	off		

prolog_load_context/2 key values

Key	Value
module	the module you are compiling into
file	absolute filename of the file being compiled
stream	the stream you are compiling from
directory	directory of the file on which the stream is open

statistics/2 memory area keywords

runtime

backtrack_free	backtrack_total
local_free	local_total
reset_free	reset_total
heap_free	heap_total
text_free	text_total
program_free	program_total
stack_free	stack_total
input_free	input_total
output_free	output_total

sttbox/2 special values

Value	Style
-1	destroy status box
65535	show the banner box, using the OEM font

stuff/3 look ahead buffer size values

Value	Compression	Window Size	Look ahead
0	9/7 bit	512 bytes	128 bytes
1	10/6 bit	1024 bytes	64 bytes
2	11/5 bit	2048 bytes	32 bytes
3	12/4 bit	4096 bytes	16 bytes
4	13/3 bit	8192 bytes	8 bytes

For most applications, a setting of 2 or 3 will produce the best compression.

system_menu/3 available functions

The File Menu options
New
Open...
Save
Save As...
Save All
Close...
Print...
Exit

The Edit Menu options
Undo
Cut
Copy
Paste
Clear
Select All

The Search Menu options
Find...
Change...
Goto Definition...
Goto Next Clause

The Run Menu options
Query...
Check Syntax
Compile
Compile All
Optimize
Optimize All
Application...

The Options Menu options
Trace
Debug
Spypoints...
Preferences...
Font...
Save Settings on Exit

The Help Menu options
Contents
How to Use Help
About WIN-PROLOG...

The File/New Option

Menu Parameter
file

Functions
new
Select the File/New option

The File/Open... Option

Menu Parameter
file

Functions
open
Select the File/Open option

open(LogFile)
Open the specified file

open(LogFile,X,Y,W,H)
Open the specified file and position its window at X,Y,W,H.

The File/Save Option

Menu Parameter
file

Functions
save
Select the File/Open option. Save the window specified in the first argument of system_menu/3. If the window is the atom untitled the contents of the untitled window are saved and the process includes the 'File/Save As...' option.

The File/Save As... Option

Menu Parameter
file

Function
save_as
Select the File/Save As... option.

save_as(LogFile)
Save the window specified in the first argument of system_menu/3 with the given file name.

The File/Save All Option

Menu Parameter
file

Function
save_all
Select the File/Save All option.

The File/Close... Option

Menu Parameter
file

Function
close
Select the File/Close... option.

The File/Print... Option

Menu Parameter
file

Function
print
Select the File/Print... option.

The File/Exit Option

Menu Parameter
file

Function
exit
Select the File/Exit option.

The Edit/Undo Option

Menu Parameter
edit

Function
undo
Select the Edit/Undo option.

The Edit/Cut Option

Menu Parameter
edit

Function
cut
Select the Edit/Cut option.

The Edit/Copy Option

Menu Parameter
edit

Function
copy
Select the Edit/Copy option.

The Edit/Paste Option

Menu Parameter
edit

Function
paste
Select the Edit/Paste option.

The Edit/Clear Option

Menu Parameter
edit

Function
clear
Select the Edit/Clear option.

The Edit/Select All Option

Menu Parameter
edit

Function
all
Select the Edit/Select All option.

The Search/Find... Option

Menu Parameter
search

Function
find
Select the Search/Find... option.

The Search/Change... Option

Menu Parameter
search

Function
change
Select the Search/Change... option.

The Search/Goto Definition... Option

Menu Parameter
search

Function
goto_definition
Select the Search/Goto Definition... option.

The Search/Goto Next Clause Option

Menu Parameter
search

Function
goto_next_clause
Select the Search/Goto Next Clause option.

The Run/Query... Option

Menu Parameter
run

Function
query
Select the Run/Query... option.

The Run/Check Syntax Option

Menu Parameter
run

Function
check_syntax
Select the Run/Check Syntax option.

The Run/Compile Option

Menu Parameter
run

Function
compile
Select the Run/Compile option.

The Run/Compile All Option

Menu Parameter
run

Function
compile_all
Select the Run/Compile All option.

The Run/Optimize Option

Menu Parameter
run

Function
optimize
Select the Run/Optimize option.

The Run/Optimize All Option

Menu Parameter
run

Function
optimize_all
Select the Run/Optimize All option.

The Run/Application... Option

Menu Parameter
run

Function
application
Select the Run/Application... option.

application(test,Main,Abort,Break,Error,Timer,Message)
Run an application in test mode with the specified hooks.
(This option is only available in the developer edition of Prolog)

application(save(File),Main,Abort,Break,Error,Timer,Message)
Save an application to the given overlay name with the specified hooks.
(This option is only available in the developer edition of Prolog)

The Options/Trace Option

Menu Parameter
options

Function
trace
Select the Options/Trace option.

The Options/Debug Option

Menu Parameter
options

Function
debug
Select the Options/Debug option.

The Options/Spypoints... Option

Menu Parameter
options

Function
spypoints
Select the Options/Spypoints... option.

The Options/Preferences... Option

Menu Parameter
options

Function
preferences
Select the Options/Preferences... option.

preferences(PM,PS,PF,SB,CV,CC,CM,CE,CR,DF,RS)
Set the Prolog preferences

The Options/Font... Option

Menu Parameter
options

Function
font
Select the Options/Font... option.

font(NewFace,NewSize,NewStyle)
Set the Prolog environment font.

The Options/Save Settings on Exit Option

Menu Parameter
options

Function
save_settings_on_exit
Select the Options/Save Settings on Exit option.

The Help/Contents Option

Menu Parameter
help

Function
contents
Select the Help/Contents option.

The Help/How to Use Help Option

Menu Parameter
help

Function
how_to_use_help
Select the Help/How to Use Help option.

The Help/About WIN-PROLOG... Option

Menu Parameter
help

Function
about_win_prolog
Select the Help/About WIN-PROLOG... option.

Style checking values

Type	Meaning
all	All the following style checking.
single_var	Checking for clauses containing a single instance of a named variable, where variables that start with a '_' are not considered named.
discontiguous	Checking for procedures whose clauses are not all adjacent to one another in the file.
multiple	Checking for multiple definitions of the same procedure in different files.

timer status values

Value	Status
0	clear the timer
Number	set the timer *Number* ticks ahead of the current time
(*Number*,*Base*)	set the timer *Number* ticks ahead of the absolute base time *Base*

timer status return values

Value	Status
0	the timer is clear
(*Interval*,*End*)	the timer was set with *Interval* and was intended to expire (or will expire) at the absolute time *End*

type/2 term type values

Value	Type
0	variable
1	integer
2	floating point number
3	true atom
4	string
5	empty list
6	list
7	tuple
8	true conjunction
9	true disjunction

wait/1 flag values

Value	Action
0	yield for one message cycle, return immediately
1	yield until at least one message is in queue

wbtnsel/2 status values

Value	Status
0	radio button deselected or checkbox unchecked
1	radio button selected or checkbox checked

wbusy/1 flag values

Value	Cursor Type
0	normal (idle) cursor
1	hourglass (busy) cursor
cursor_handle	cursor set to the specified (busy) cursor

wedtclp/2 clipboard function values

Value	Function
1	Cut
2	Copy
3	Paste
4	Clear
5	Undo

Negative values for these functions perform tests to see whether the functions would succeed.

Available control window classes

Name	Type
button	push buttons, radio buttons, checkboxes
combobox	edit control with single choice list box
edit	edit control (the edit space is allocated from the local heap and is therefore limited)
editor	special 30,000 byte edit control (30,000 bytes is allocated for each editor control from the global heap)
listbox	single or multiple choice list box
scrollbar	scroll bar
static	static text or icon window
grafix	graphical control window

Window class names

Name	Type
`Button`	push buttons, radio buttons and checkboxes
`Edit`	edit fields
`ListBox`	single and multiple choice listboxes
`ComboBox`	drop-down comboboxes and edimboboxes
`Static`	static text fields or icon fields
`ScrollBar`	stand-alone scrollbars
`P386Main`	Prolog's main application window
`MDIClient`	Prolog's MDI client area
`P386Term`	Prolog's console window

Predefined Windows classes

Class	Description
button	push buttons, radio buttons, checkboxes
combobox	edit control with single choice list box
edit	all types of edit control
listbox	single or multiple choice list box
scrollbar	scroll bar
static	static text or icon window

LPA defined window classes

Class	Description
text	mdi child with 30,000 byte text control
user	mdi child window with no controls
dialog	top level dialog with no controls
editor	special 30,000 byte edit control
grafix	special graphics input/output window class

wenable/2 window status values

Value	Action
0	disable the window
1	enable the window

wfcreate/4 font style values

Value	Style
0	normal
1	italic
2	bold
3	bold/italic

wflag/1 interrupt flag values

Value	Meaning
0	Windows interrupts disabled
1	Windows interrupts enabled

Special numeric font names

Value	Type
0	fixed IBM PC (OEM) font
1	proportional Windows (SYSTEM) font
2	fixed Windows (SYSTEM FIXED) font
3	proportional ANSI (ANSI VAR) font
4	fixed ANSI (ANSI FIXED) font
5	device default (DEVICE DEFAULT) font

wgfx/2 Windows graphics functions

Function	Action
line(x1,y1,x2,y2,..xn,yn)	draw a polyline
poly(x1,y1,x2,y2,..xn,yn)	draw a polygon
rect(x1,y1,x2,y2)	draw a rectangle
elip(x1,y1,x2,y2)	draw an ellipse
box(x1,y1,x2,y2,xd,yd)	draw a rounded rectangle
arc(x1,y1,x2,y2,xs,ys,xf,yf)	draw an arc
seg(x1,y1,x2,y2,xs,ys,xf,yf)	draw a segment
pie(x1,y1,x2,y2,xs,ys,xf,yf)	draw a pie
text(x1,y1,string)	draw a text string
fill(x1,y,red,grn,blu)	perform a flood fill
brsh(red,grn,blu,type)	create and select a brush
pen(red,grn,blu,type)	create and select a pen
fore(red,grn,blu)	set text foreground colour
back(red,grn,blu)	set text background colour
mode(mode)	set drawing mode
trns(mode)	set text background mode
font(font)	select a text font
org(x0,y0)	set window origin
meta(x1,y1,x2,y2,meta)	play a metafile
icon(x1,y1,icon)	draw an icon
bits(x1,y1,x2,y2,x0,y0,bitmap)	draw a bitmap

Predefined window cursors

Value	Cursor Type
0	Default cursor
1	Busy cursor
2	Normal arrow
3	I-Beam
4	Hourglass
5	Crosshair
6	Vertical arrow
7	Size cursor
8	Icon cursor
9	Northwest/Southeast cursor
10	North/South cursor
11	West/East cursor
12	Northeast/Southwest cursor

Graphics instructions used to detect hits with wgfxtst/5

Graphics Instruction	Function
poly/n	draw a polygon
rect/4	draw a rectangle
elip/4	draw an ellipse
box/6	draw a rounded rectangle
text/3	draw a text string
icon/3	draw an icon
bits/7	draw a bitmap

Graphics instructions that affect wgfxtst/5

Graphics Instruction	Function
font/1	select a text font
org/2	set window origin

winapi/5 function type casts

Name	Description
byte	function returns an 8-bit integer
word	function returns a 16-bit integer
long	function returns a 32-bit integer
text	function returns a string pointer

winapi/5 parameter type casts

Name	Description
byte	an 8-bit integer is pushed onto the stack
word	a 16-bit integer is pushed onto the stack
long	a 32-bit integer is pushed onto the stack
text	a string pointer is pushed onto the stack
[..]	a structure address is pushed onto the stack

winapi/5 structure element type casts

Name	Description
byte	an 8-bit integer is stored in the structure
word	a 16-bit integer is stored in the structure
long	a 32-bit integer is stored in the structure
text	a string pointer is stored in the structure

wlbxsel/3 listbox selection state values

Value	Selection State
0	item is not selected
1	item is selected

wlink/3 relation values

Value	Relation
-1	returns parent of window
0	returns first sibling of window
1	returns last sibling of window
2	returns next sibling of window
3	returns previous sibling of window
4	returns owner of window
5	returns first child of window

wmnunbl/3 enable status values

Value	Status
0	disable the item
1	enable the item
2	disable the item without greying

wmnusel/3 menu selection values

Value	Status
0	item is not ticked
1	item is ticked

wprnend/1 terminate code values

Value	Action
0	Finish printing normally
1	Abort printing abruptly

wprnstt/1 printer status values

Value	Status
0	Printer not initialised
1	Printer initialised, no page
2	Printer at start of fresh page
3	Printer at work on current page

wrange/4 scrollbar type values

Value	Type
0	Window is a scroll bar, address directly
1	Address horizontal scroll bar of window
2	Address vertical scroll bar of window

wshow/2 window visibility status values

Value	Status
0	hide the window
1	normalise the window
2	minimise the window
3	maximise the window

wthumb/3 scrollbar type values

Value	Type
0	Window is a scroll bar, address directly
1	Address horizontal scroll bar of window
2	Address vertical scroll bar of window

The standard ordering of terms

Type	Order
variables	are less than:
integers and floats	which are less than:
atoms	which are less than:
strings	which are less than:
lists	which are less than:
compound terms	which are less than:
true conjunctions	which are less than:
true disjunctions	

Type comparisons

Type	Comparison
variables	address
integers and floats	numerical value
atoms	ascii value
strings	ascii value

Window Styles

Generic window styles

ws_popup	ws_child	ws_clipsiblings	ws_clipchildren
ws_visible	ws_disabled	ws_minimize	ws_maximize
ws_caption	ws_border	ws_vscroll	ws_hscroll
ws_sysmenu	ws_thickframe	ws_minimizebox	ws_maximizebox
ws_group	ws_tabstop		

Button window styles

bs_pushbutton	bs_defpushbutton	bs_checkbox	bs_autocheckbox
bs_radiobutton	bs_3state	bs_auto3state	bs_groupbox
bs_autoradiobutton	bs_ownerdraw	bs_lefttext	

Edit window styles

es_left	es_center	es_right	es_multiline
es_uppercase	es_lowercase	es_password	es_autovscroll
es_autohscroll	es_nohidesel	es_oemconvert	es_readonly
es_wantreturn			

List box window styles

lbs_notify	lbs_sort	lbs_noredraw	lbs_multiplesel
lbs_ownerdrawfixed	lbs_ownerdrawvariable	lbs_hasstrings	lbs_usetabstops
lbs_nointegralheight	lbs_multicolumn	lbs_wantkeyboardinput	lbs_extendedsel
lbs_disablenoscroll			

Combo box window styles

cbs_simple	cbs_dropdown	cbs_dropdownlist	cbs_ownerdrawfixed
cbs_ownerdrawvariable	cbs_autohscroll	cbs_oemconvert	cbs_sort
cbs_hasstrings	cbs_nointegralheight	cbs_disablenoscroll	

Static window styles

ss_left	ss_center	ss_right	ss_icon
ss_blackrect	ss_grayrect	ss_whiterect	ss_blackframe
ss_grayframe	ss_whiteframe	ss_simple	ss_leftnowordwrap
ss_noprefix			

Scroll bar window styles

sbs_horz	sbs_vert	sbs_topalign	sbs_leftalign
sbs_bottomalign	sbs_rightalign		

Dialog window styles

dlg_ownedbydesktop	dlg_ownedbyprolog

Generic window styles

ws_popup	ws_child	ws_clipsiblings	ws_clipchildren
ws_visible	ws_disabled	ws_minimize	ws_maximize
ws_caption	ws_border	ws_vscroll	ws_hscroll
ws_sysmenu	ws_thickframe	ws_minimizebox	ws_maximizebox
ws_group	ws_tabstop		

ws_popup

Value	Window Type	Description
16'80000000	desktop window	Must be applied to windows that do not have parents

ws_child

Value	Window Type	Description
16'40000000	child window	Must be applied to windows that have parents.

ws_clipsiblings

Value	Window Type	Description
16'04000000	child window	Prevent neighbouring sibling windows from being drawn over during the handling of a paint message.

ws_clipchildren

Value	Window Type	Description
16'02000000	parent window	Prevent child windows from being drawn over during the handling of a paint message.

ws_visible

Value	Window Type	Description
16'10000000	any type	Create the window visibly (with respect to the parent window).

ws_disabled

Value	Window Type	Description
16'08000000	any type	Create the window initially disabled.

ws_minimize

Value	Window Type	Description
16'20000000	any type	Create the window minimized.

ws_maximize

Value	Window Type	Description
16'01000000	any type	Create the window maximized.

ws_caption

Value	Window Type	Description
16'00C00000	any type	Create the window with a title bar.

ws_border

Value	Window Type	Description
16'00800000	any type	Create the window with a border.

ws_vscroll

Value	Window Type	Description
16'00200000	any type	Create the window with a vertical scroll bar.

ws_hscroll

Value	Window Type	Description
16'00100000	any type	Create the window with a horizontal scroll bar.

ws_sysmenu

Value	Window Type	Description
16'00080000	any type	Create the window with a system menu.

ws_thickframe

Value	Window Type	Description
16'00040000	any type	Create the window with a thick frame that can be used to size the window

ws_minimizebox

Value	Window Type	Description
16'00020000	parent windows	Create the window with a minimize button.

ws_maximizebox

Value	Window Type	Description
16'00010000	parent windows	Create the window with a maximize button.

ws_group

Value	Window Type	Description
16'00020000	dialog control windows	Specify the control as the first in a group. You can move between controls in a group using the direction keys. The next control defined with the ws_group style ends this group and starts a new one.

ws_tabstop

Value	Window Type	Description
16'00010000	dialog control windows	Allow the window to be selected using the tab key.

Button window styles

bs_pushbutton
bs_defpushbutton
bs_checkbox
bs_autocheckbox
bs_radiobutton
bs_3state
bs_auto3state
bs_groupbox
bs_autoradiobutton
bs_ownerdraw
bs_lefttext

bs_pushbutton

Value	Window Type	Description
16'00000000	button	Specifies a push button that sends a msg_button when clicked. This is the default style.

bs_defpushbutton

Value	Window Type	Description
16'00000001	button	Gives the button a bold border. The button represents the default response by the user.

bs_checkbox

Value	Window Type	Description
16'00000002	button	Specifies a check box button. This style overrides the bs_pushbutton style.

bs_autocheckbox

Value	Window Type	Description
16'00000003	button	Same as the bs_checkbox style, except that the button automatically toggles its state when clicked on by the user.

bs_3state

Value	Window Type	Description
16'00000005	button	Same as the bs_checkbox style, except that the button can be greyed as well as checked.

bs_auto3state

Value	Window Type	Description
16'00000006	button	Same as the bs_3state style, except that the toggling between the three states is handled automatically.

bs_groupbox

Value	Window Type	Description
16'00000007	button	Specifies a box into which other buttons may be grouped. The text for the button appears in the top-left corner.

bs_radiobutton

Value	Window Type	Description
16'00000004	button	Specifies a radio button. This style overrides the bs_pushbutton style.

bs_autoradiobutton

Value	Window Type	Description
16'00000009	button	Identical to bs_radiobutton, except that all other radio buttons in the same group are deselected automatically.

bs_ownerdraw

Value	Window Type	Description
16'0000000B	button	Specifies a button where the drawing of the button is down to the user.

bs_lefttext

Value	Window Type	Description

16'00000020	button	Causes the text always appear on the left for a radio button or checkbox button. This is used with the bs_checkbox, bs_radiobutton or bs_3state styles.

Edit window styles

es_left
es_center
es_right
es_multiline
es_uppercase
es_lowercase
es_password
es_autovscroll
es_autohscroll
es_nohidesel
es_oemconvert
es_readonly
es_wantreturn

es_left

Value	Window Type	Description
16'00000000	edit	Align text on the left hand side.

es_center

Value	Window Type	Description
16'00000001	edit	Center text in an edit control that has the es_multiline style.

es_right

Value	Window Type	Description
16'00000002	edit	Align text on the right hand side in an edit control with the es_multiline style.

es_multiline

Value	Window Type	Description
16'00000004	edit	Specifies an edit control that can contain more than one line of text. Used with es_autohscroll and es_autovscroll.

es_autovscroll

Value	Window Type	Description
16'00000040	edit	Automatically scrolls text up one page when the user presses the <enter> key on the last line. Used with es_wantreturn and es_multiline. If es_autovscroll is not set and the edit control has the es_multiline style then the edit control word shows as many lines as it can and beeps when no more lines can be displayed.

es_autohscroll

Value	Window Type	Description
16'00000080	edit	Automatically scrolls to the right by 10 characters when the user types a character at the end of a line. Used with es_wantreturn and es_multiline. If es_autohscroll is not set and the edit control has the es_multiline style then the edit control word wraps at the window boundary.

es_uppercase

Value	Window Type	Description
16'00000008	edit	Converts all characters into uppercase as they are typed into the control.

es_lowercase

Value	Window Type	Description
16'00000010	edit	Converts all characters into lowercase as they are typed into the control.

es_password

Value	Window Type	Description
16'00000020	edit	Displays all characters as an asterisk as they are typed into the control.

es_nohidesel

Value	Window Type	Description
16'00000100	edit	Normally, an edit control hides the current selection when the control loses focus. Adding this style stops this behaviour.

es_oemconvert

Value	Window Type	Description
16'00000400	edit	Any text entered in the edit control is immediately converted from ANSI into OEM and then back to ANSI. This ensures that a correct conversion will take place if ansoem/2 is used on the text in the edit control.

es_readonly

Value	Window Type	Description
16'00000800	edit	Sets the edit control as a read-only edit control.

es_wantreturn

Value	Window Type	Description
16'00001000	edit	When the user hits the <return> key and an edit control with this style is in focus, the return goes into the edit control rather than being processed by the parent of the edit control.

Listbox window styles

lbs_notify
lbs_sort
lbs_noredraw
lbs_multiplesel
lbs_ownerdrawfixed
lbs_ownerdrawvariable
lbs_hasstrings
lbs_usetabstops
lbs_nointegralheight
lbs_multicolumn
lbs_wantkeyboardinput
lbs_extendedsel
lbs_disablenoscroll

lbs_notify

Value	Window Type	Description
16'00000001	listbox	The parent window recieves a message whenever a selection is made in the list box. If the selection is made with the keyboard or a single mouse click then the msg_select message is sent. If the selection is made with a double mouse click then the msg_double message is sent.

lbs_sort

Value	Window Type	Description
16'00000002	listbox	The strings in the list box are sorted alphabetically.

lbs_noredraw

Value	Window Type	Description
16'00000004	listbox	The list box display is not updated when changes are made.

lbs_multiplesel

Value	Window Type	Description
16'00000008	listbox	Multiple selections are allowed. Clicking on a string toggles its selection status.

lbs_ownerdrawfixed

Value	Window Type	Description
16'00000010	listbox	The owner of the list box is responsible for drawing its contents.

lbs_ownerdrawvariable

Value	Window Type	Description
16'00000020	listbox	The owner of the list box is responsible for drawing its contents. The items in the listbox may be variable in height.

lbs_hasstrings

Value	Window Type	Description
16'00000040	listbox	Sets an owner-draw list box which contains items consisting of strings.

lbs_usetabstops

Value	Window Type	Description
16'00000080	listbox	Turn on the expansion of tab characters in the list box items.

lbs_nointegralheight

Value	Window Type	Description
16'00000100	listbox	Normally Windows adjusts the vertical size of list boxes so that only whole lines are displayed. Applying this style to a list box ensures that it has the exact height you specify.

lbs_multicolumn

Value	Window Type	Description
16'00000200	listbox	Sets a multi-column window that scrolls horizontally.

lbs_wantkeyboardinput

Value	Window Type	Description
16'00000400	listbox	This style has no effect in LPA Prolog for Windows.

lbs_extendedsel

Value	Window Type	Description
16'00000800	listbox	Allows the user to select multiple items in the list box using the <shift> key and the mouse or other special key combinations.

lbs_disablenoscroll

Value	Window Type	Description
16'00001000	listbox	Specifies a list box that has a scroll bar regardless of the number of elements it contains.

Combo box window styles

cbs_simple
cbs_dropdown
cbs_dropdownlist
cbs_ownerdrawfixed
cbs_ownerdrawvariable
cbs_autohscroll
cbs_oemconvert
cbs_sort
cbs_hasstrings
cbs_nointegralheight
cbs_disablenoscroll

cbs_simple

Value	Window Type	Description
16'00000001	combobox	The list box component of the combo box is always shown and the current selection appears in the edit box component

cbs_dropdown

Value	Window Type	Description
16'00000002	combobox	Similar to the cbs_simple style except that the list box component of the combo box is only displayed when the user selects the icon next to the edit box component.

cbs_dropdownlist

Value	Window Type	Description
16'00000003	combobox	Similar to the cbs_dropdown style except that the selection field, which is normally an edit control, in this case is a static text control.

cbs_ownerdrawfixed

Value	Window Type	Description
16'00000010	combobox	The owner of the list box is responsible for drawing its contents.

cbs_ownerdrawvariable

Value	Window Type	Description
16'00000020	combobox	The owner of the list box is responsible for drawing its contents. The items in the listbox may be variable in height.

cbs_autohscroll

Value	Window Type	Description
16'00000040	combobox	When this style is set, the text in the edit control component of the combo box is automatically scrolled to the right when a character is typed in at the right hand edge of the control. If this style is not set only text that fits in the edit control is allowed.

cbs_oemconvert

Value	Window Type	Description
16'00000080	combobox	Any text entered in the edit control is immediately converted from ANSI into OEM and then back to ANSI This ensures that a correct conversion will take place if ansoem/2 is used on the text in the edit control. This applies only to combo boxes that have the cbs_simple or cbs_dropdown styles.

cbs_sort

Value	Window Type	Description
16'00000100	combobox	Strings entered into the list box component of the combo box are automatically sorted.

cbs_hasstrings

Value	Window Type	Description
16'00000200	combobox	An owner-draw combo box which contains items

consisting of strings.

cbs_nointegralheight

Value	Window Type	Description
16'00000400	combobox	Normally Windows adjusts the vertical size of combo boxes so that only whole lines are displayed. Applying this style to a combo box ensures that it has the exact height you specify.

cbs_disablenoscroll

Value	Window Type	Description
16'00000800	combobox	Specifies a combo box that has a scroll bar regardless of the number of elements it contains.

Static window styles

ss_left
ss_center
ss_right
ss_icon
ss_blackrect
ss_grayrect
ss_whiterect
ss_blackframe
ss_grayframe
ss_whiteframe
ss_simple
ss_leftnowordwrap
ss_noprefix

ss_left

Value	Window Type	Description
16'00000000	static	Displays text flush-left in the static text control.

ss_center

Value	Window Type	Description
16'00000001	static	Displays text centered in the static text control.

ss_right

Value	Window Type	Description
16'00000002	static	Displays text flush-right in the static text control.

ss_icon

Value	Window Type	Description
16'00000003	static	Sets the static control to contain an icon whose name is given as the text of the static control.

ss_blackrect

Value	Window Type	Description
16'00000004	static	Specifies a rectangle filled with the colour used to draw window frames.

ss_grayrect

Value	Window Type	Description
16'00000005	static	Specifies a rectangle filled with the colour used to fill the screen background.

ss_whiterect

Value	Window Type	Description
16'00000006	static	Specifies a rectangle filled with the colour used to fill window backgrounds.

ss_blackframe

Value	Window Type	Description
16'00000007	static	Specifies a rectangle bordered with the colour used to draw window frames.

ss_grayframe

Value	Window Type	Description
16'00000008	static	Specifies a rectangle bordered with the colour used to fill the screen background.

ss_whiteframe

Value	Window Type	Description
16'00000009	static	Specifies a rectangle bordered with the colour used to fill window backgrounds.

ss_simple

Value	Window Type	Description
16'0000000B	static	Specifies a static control that contains a single-line of text that is displayed flush-left.

ss_leftnowordwrap

Value	Window Type	Description
16'0000000C	static	Specifies a static control that displays the given text

flush-left and may contain multiple lines. Any text that exceeds the width of the control will be clipped and not word-wrapped.

ss_noprefix

Value	Window Type	Description
16'00000080	static	Normally Windows interperets the "&" character in the text of a static field as denoting that the character following the ampersand is an accelerator. This style specifies an edit control that does not interperet the "&" character.

Scroll bar window styles

sbs_horz
sbs_vert
sbs_topalign
sbs_leftalign
sbs_bottomalign
sbs_rightalign

sbs_horz

Value	Window Type	Description
16'00000000	scrollbar	Specifies a horizontal scroll bar.

sbs_vert

Value	Window Type	Description
16'00000001	scrollbar	Specifies a vertical scroll bar.

sbs_topalign

Value	Window Type	Description
16'00000002	scrollbar	Used with the sbs_horz style. The top edge of the scroll bar is alligned with the top of the rectangle specifying its size. The scroll bar has the default height for system scrollbars.

sbs_leftalign

Value	Window Type	Description
16'00000002	scrollbar	Used with the sbs_vert style. The left edge of the scroll bar is alligned with the left of the rectangle specifying its size. The scroll bar has the default width for system scrollbars.

sbs_bottomalign

Value	Window Type	Description
16'00000004	scrollbar	Used with the sbs_horz style. The bottom edge of the scroll bar is alligned with the bottom of the rectangle specifying its size. The scroll bar has the default height for system scrollbars.

sbs_rightalign

Value	Window Type	Description
16'00000004	scrollbar	Used with the sbs_vert style. The right edge of the scroll bar is alligned with the right of the rectangle specifying its size. The scroll bar has the default width for system scrollbars.

Dialog window styles

dlg_ownedbydesktop
dlg_ownedbyprolog

dlg_ownedbydesktop

Value	Window Type	Description
16'00000000	dialog	This default style for dialogs sets the desktop as the owner of dialogs. This means that the dialog will appear independently of Prolog and can be hidden behind Prolog's main application window.

dlg_ownedbyprolog

Value	Window Type	Description
16'00000001	dialog	Used to set Prolog as the owner of dialogs. This means that the dialog will always appear on top of Prolog's main application window.

General message names

msg_menu	msg_sysmenu	msg_close	msg_focus
msg_fuzzy	msg_change	msg_button	msg_select
msg_double	msg_size	msg_move	msg_horz
msg_vert	msg_paint	msg_leftdown	msg_leftdouble
msg_leftup	msg_rightdown	msg_rightdouble	msg_rightup
msg_mousemove	msg_char		

msg_menu

Data	Description
menu item ID	A menu item has been selected.

The data parameter is an integer giving the ID of the menu item selected.

msg_sysmenu

Data	Description
menu item ID	A menu item has been selected from the system menu.

The data parameter is an integer giving the ID of the menu item selected.

msg_close

Data	Description
no data	Window has been requested to close

msg_focus

Data	Description
no data	Window has come into focus

msg_fuzzy

Data	Description
no data	Window has gone out of focus

msg_change

Data	Description
edit operation flag	Edit or editor window has been changed

The msg_change data parameter can have the following values:

Value	Change Action	
0	no operation.	The change operation did not affect the text of the window
1	success.	The text in the window was changed successfully
2	truncation.	The text in the window was changed but output was truncated

Certain actions that cause change messages in a text window do not affect the text of the window. These are: changing the selection range, positioning the cursor at the beginning or end of a text window using the <ctrl>-home or <ctrl>-end keys and copying text.

msg_button

Data	Description
no data	Button has been pressed

msg_select

Data	Description
no data	List or Combo box selection made

msg_double

Data	Description
no data	List or Combo box double clicked

msg_size

Data	Description
no data	Window has been sized

msg_move

Data	Description
no data	Window has been moved

msg_horz

Data	Description
new scroll position	Horizontal scroll bar been moved.

The data parameter is an integer showing the new scroll position.

msg_vert

Data	Description
new scroll position	Vertical scroll bar been moved.

The data parameter is an integer showing the new scroll position.

msg_paint

Data	Description
window type	Window needs repainting.

The msg_paint data parameter can have the following values:

Value	Window Type
grafix	grafix window
button_up	unpressed button
button_down	pressed button

msg_leftdown

Data	Description
mouse coordinates	Left mouse button has been pressed in either the main window (while the *wbusy/1* flag is set greater than 0) or a "grafix" window.

The data is a coordinate pair of the form:

```
(X,Y)
```

showing the position of the mouse click.

msg_leftdouble

Data	Description
mouse coordinates	Left mouse button has been double-clicked in a "grafix" window.

The data is a coordinate pair of the form:

```
(X,Y)
```

showing the position of the mouse click.

msg_leftup

Data	Description
mouse coordinates	Left mouse button has been released in a "grafix" window.

The data is a coordinate pair of the form:

```
(X,Y)
```

showing the position of the mouse release.

msg_rightdown

Data	Description
mouse coordinates	Right mouse button has been pressed in either the main window (while the *wbusy/1* flag is set greater than 0) or a "grafix" window.

The data is a coordinate pair of the form:

```
(X,Y)
```

showing the position of the mouse click.

msg_rightdouble

Data	Description
mouse coordinates	Right mouse button has been double-clicked in a "grafix" window.

The data is a coordinate pair of the form:

```
(X,Y)
```

showing the position of the mouse click.

msg_rightup

Data	Description
mouse coordinates	Right mouse button has been released in a "grafix" window.

The data is a coordinate pair of the form:

```
(X,Y)
```

showing the position of the mouse release.

msg_mousemove

Data	Description
mouse coordinates	Mouse pointer has been moved in a "grafix" window.

The data is a coordinate pair of the form:

```
(X,Y)
```

showing the current position of the mouse.

msg_char

Data	Description
key code	Character returned from the keyboard in a "grafix" window.

The data is either the ASCII code of the key that was pressed, or one of the following atoms:

Key Atom Names
prior
next
end
home
left
up
right
down
select
print
execute
snapshot
insert
delete

Integer message and data values

msg_menu msg_sysmenu msg_close msg_focus
msg_fuzzy msg_change msg_button msg_select
msg_double msg_size msg_move msg_horz
msg_vert msg_paint msg_leftdown msg_leftdouble
msg_leftup msg_rightdown msg_rightdouble msg_rightup
msg_mousemove msg_char

msg_menu Integer Value 0

Data	Description
menu item ID	A menu item has been selected.

The data parameter is an integer giving the ID of the menu item selected.

msg_sysmenu Integer Value 1

Data	Description
menu item ID	A menu item has been selected from the system menu.

The data parameter is an integer giving the ID of the menu item selected.

msg_close Integer Value 2

Data	Description
no data	Window has been requested to close

msg_focus Integer Value 3

Data	Description
no data	Window has come into focus

msg_fuzzy Integer Value 4

Data	Description
no data	Window has gone out of focus

msg_change Integer Value 5

Data	Description
no data	Edit or editor window has been changed

msg_button Integer Value 6

Data	Description
no data	Button has been pressed

msg_select Integer Value 7

Data	Description
no data	List or Combo box selection made

msg_double Integer Value 8

Data	Description
no data	List or Combo box double clicked

msg_size Integer Value 9

Data	Description
no data	Window has been sized

msg_move Integer Value 10

Data	Description
no data	Window has been moved

msg_horz Integer Value 11

Data	Description
new scroll position	Horizontal scroll bar been moved.

The data parameter is an integer showing the new scroll position.

msg_vert Integer Value 12

Data	Description
new scroll position	Vertical scroll bar been moved.

The data parameter is an integer showing the new scroll position.

msg_paint Integer Value 13

Data	Description
window type	Window needs repainting.

The msg_paint data parameter can have the following values:

Value	Window Type
0	grafix window
1	unpressed button
2	pressed button

msg_leftdown Integer Value 14

Data	Description
mouse coordinates	Left mouse button has been pressed in either the main window (while the *wbusy/1* flag is set greater than 0) or a "grafix" window.

The data parameter is a 32-bit integer that contains the X and Y coordinates of the mouse position, relative to the window origin, in the low and high words respectively.

msg_leftdouble Integer Value 15

Data	Description
mouse coordinates	Left mouse button has been double-clicked in a "grafix" window.

The data parameter is a 32-bit integer that contains the X and Y coordinates of the mouse position, relative to the window origin, in the low and high words respectively.

msg_leftup Integer Value 16

Data	Description
mouse coordinates	Left mouse button has been released in a "grafix" window.

The data parameter is a 32-bit integer that contains the X and Y coordinates of the mouse position, relative to the window origin, in the low and high words respectively.

msg_rightdown Integer Value 17

Data	Description
mouse coordinates	Right mouse button has been pressed in either the main window (while the *wbusy/1* flag is set greater than 0) or a "grafix" window.

The data parameter is a 32-bit integer that contains the X and Y coordinates of the mouse position, relative to the window origin, in the low and high words respectively.

msg_rightdouble Integer Value 18

Data	Description
mouse coordinates	Right mouse button has been double-clicked in a "grafix" window.

The data parameter is a 32-bit integer that contains the X and Y coordinates of the mouse position, relative to the window origin, in the low and high words respectively.

msg_rightup Integer Value 19

Data	Description
mouse coordinates	Right mouse button has been released in a "grafix" window.

The data parameter is a 32-bit integer that contains the X and Y coordinates of the mouse position, relative to the window origin, in the low and high words respectively.

msg_mousemove Integer Value 20

Data	Description
mouse coordinates	Mouse pointer has been moved in a "grafix" window.

The data parameter is a 32-bit integer that contains the X and Y coordinates of the mouse position, relative to the window origin, in the low and high words respectively.

msg_char Integer Value 21

Data	Description
key code	Character returned from the keyboard in a "grafix" window.

The data is either the ASCII code of the key that was pressed, or one of the following codes:

Special Key Values

Code	Key
1	"Prior"
2	"Next"
3	"End"
4	"Home"
5	"Left"
6	"Up"
7	"Right"
8	"Down"
9	"Select"
10	"Print"
11	"Execute"
12	"Snapshot"
13	"Insert"
14	"Delete"

Find box message names

Name	Value	Description
msg_fbclose	70	The Find Box close button
msg_fbfind	71	The Find Box Find button
msg_fbfndnxt	72	The Find Box Find next button

Change box message names

Name	Value	Description
msg_cbclose	80	The Change Box close button
msg_cbfind	81	The Change Box Find button
msg_cbfndnxt	82	The Change Box Find next button
msg_cbchange	83	The Change Box Change button
msg_cbchgfnd	84	The Change Box Change+find button
msg_cbchgall	85	The Change Box Change all button

Status box message names

Name	Value	Description
msg_sbcancel	90	The Status Box Cancel button

Prolog window handle types

Argument	Type	Meaning
Window	<atom>	a named top-level *Window* created by Prolog.
(Window,ID)	(<atom>,<integer>)	a control item with the given ID within the named top-level Window.
Handle	<integer>	the *Handle* of any window (not just those created by Prolog)
(*Handle*,*ID*)	(<integer>,<integer>)	a control item with the given *ID* that is a child of the window with the given *Handle*

Prolog's special numeric window handles

Handle	Window
0	the main (parent) Prolog window
1	the console (user input/output) window
2	the status box window

Prolog's special numeric window IDs

ID	Window
0	the window itself
1	the edit control of an MDI child window.

Prolog's special numeric font numbers

Value	Font
0	OEM font (IBM PC font, fixed pitch, OEM character set)
1	ANSI font (Windows font, variable pitch, ANSI character set)

Programmable hook names and their built-in equivalents

Default Hook	Built-in Equivalent
'?ERROR?'/2	error_hook/2
'?BREAK?'/1	break_hook/1
'?DEBUG?'/1	debug_hook/1
'?ABORT?'/0	abort_hook/0
'?KEY?'/2	key_hook/2
'?KEY?'/3	key_hook/3
'?CHANGE?'/3	change_hook/3
'?FIND?'/3	find_hook/3
'?DLL?'/3	dll_hook/3
'?MESSAGE?'/4	message_hook/4

Dialog and menu handlers

Function	Functor/Arity	Built-in
modal dialogs	<user defined>/4	modal_handler/4
modeless dialogs	<user defined>/3	modeless_handler/3
menus	<user defined>/2	menu_handler/2

Control keys with pre-defined functions

Control Key	Function
<ctrl-A>	Select All
<ctrl-C>	Copy
<ctrl-H>	Back Space
<ctrl-I>	Tab
<ctrl-J>	New Line
<ctrl-M>	Carriage Return
<ctrl-V>	Paste
<ctrl-X>	Cut
<ctrl-Z>	Undo
<ctrl-[>	End Of File

The possible values of the "syntax_errors" prolog flag

Value	Behaviour
quiet	When a syntax error is detected, nothing is printed, and *read/1* just quietly fails.
dec10	When a syntax error is detected, a syntax error message is printed, and the read is repeated.
fail	When a syntax error is detected, a syntax error is printed and the read then fails.
error	When a syntax error is detected, an error is thrown to the error handler.

Command-Line Switches

Prolog command line switches allow you to set aspects of Prolog's configuration at the command line. The following command-line switches are available

Switch	Meaning	Default Value
/B	backtrack stack	(default 64)
/L	local stack	(default 64)
/R	reset stack	(default 64)
/H	heap space	(default 256)
/T	text space	(default 512)
/P	program space	(default 2048)
/S	system stack	(default 64)
/I	string input space	(default 64)
/O	string output space	(default 64)
/N	initialisation file	(default 0)
/V	banner flag	(default 0)
/Z	3d-look dialogs	(default 0)

/B - The backtrack stack switch

The size of the backtrack stack governs the number of choice points allowed during an evaluation. To run Prolog with your own setting for the backtrack stack you can set the /B backtrack stack switch at the Prolog start-up command line. For example, you can set the backtrack stack of Prolog to have a size of 64k by entering the following at the DOS prompt:

```
C> WIN PRO386W /B64
```

You can also configure the backtrack stack by setting the /b command line switch on the command line property of Prologs program item.

/I - The string input space switch

Two additional string input and output buffers have been added to Prolog. Normally these default to 64k per buffer but you can set them to any value that is appropriate to the amount of memory available on your machine. For example, you can set the string input buffer of Prolog to have a size of 1Mb by entering the following at the DOS prompt:

```
C> WIN PRO386W /I1000
```

You can also configure the string input buffer by setting the /I command line switch on the command line property of Prologs program item.

/H - The heap space switch

The size of the heap space governs the number or size of data structures allowed during an evaluation. To run Prolog with your own setting for the heap space you can set the /H heap space switch at the Prolog start-up command line. For example, you can set the heap space of Prolog to have a size of 1024k by entering the following at the DOS prompt:

```
C> WIN PRO386W /H1024
```

You can also configure the backtrack stack by setting the /H command line switch on the command line property of Prologs program item.

/L - The local stack switch

The size of the backtrack stack governs the depth of non-tail recursive calls allowed during an evaluation. To run Prolog with your own setting for the backtrack stack you can set the /L local stack switch at the Prolog start-up command line. For example, you can set the local stack of Prolog to have a size of 64k by entering the following at the DOS prompt:

```
C> WIN PRO386W /L64
```

You can also configure the local stack by setting the /L command line switch on the command line property of Prologs program item.

/N - The initialisation file switch

This switch allows you to switch the location of the PRO386W.INI file between one of three possible locations, or not to use this file at all, to overcome the problems some users have experienced on networks. The /N switch has the following meanings:

Switch	Location
/N0	Use Prolog directory (default)
/N1	Use Windows directory
/N2	Use Current directory
/N3	Do not save .INI file

For example, you can set Prolog to use the Windows directory as the location for the initialisation file by entering the following at the DOS prompt:

```
C> WIN PRO386W /N1
```

You can also configure the initialisation file by setting the /N command line switch on the command line property of Prologs program item.

/O - The string output space switch

Two additional string input and output buffers have been added to Prolog. Normally these default to 64k per buffer but you can set them to any value that is appropriate to the amount of memory available on your machine. For example, you can set the string output buffer of Prolog to have a size of 1Mb by entering the following at the DOS prompt:

```
C> WIN PRO386W /O1000
```

You can also configure the string output buffer by setting the /O command line switch on the command line property of Prologs program item.

/P - The program space switch

The size of the program space governs the size of program allowed in Prolog. To run Prolog with your own setting for the program space you can set the /P program space switch at the Prolog startup command line. For example, you can set the program space of Prolog to have a size of 4096k by entering the following at the DOS prompt:

```
C> WIN PRO386W /P4096
```

You can also configure the program space by setting the /P command line switch on the command line property of Prologs program item.

/R - The reset stack switch

The size of the reset stack governs the number of variables allowed that might need unbinding during an evaluation. To run Prolog with your own setting for the reset stack you can set the /R reset stack switch at the Prolog start-up command line. For example, you can set the reset stack of Prolog to have a size of 128k by entering the following at the DOS prompt:

```
C> WIN PRO386W /R128
```

You can also configure the reset stack by setting the /R command line switch on the command line property of Prologs program item.

/S - The system stack switch

The size of the system stack governs the maximum depth of structures that can be held in memory. To run Prolog with your own setting for the system stack you can set the /S system stack switch at the Prolog start-up command line. For example, you can set the system stack of Prolog to have a size of 128k by entering the following at the DOS prompt:

```
C> WIN PRO386W /S128
```

You can also configure the system stack by setting the /S command line switch on the command line property of Prologs program item.

/T - The text space switch

The size of the text space governs the number atoms or strings created during an evaluation. To run Prolog with your own setting for the text space you can set the /T text space switch at the Prolog start-up command line. For example, you can set the text space of Prolog to have a size of 1024k by entering the following at the DOS prompt:

```
C> WIN PRO386W /T1024
```

You can also configure the text space by setting the /T command line switch on the command line property of Prologs program item.

/V - The banner flag switch

Another command line switch lets you can decide whether or not to suppress the welcome banner. The /V switch can take either of the following values:

Switch	Meaning
/V0	show the welcome banner
/V1	hide the welcome banner

For example, you can set Prolog to start up without displaying the welcome banner by entering the following at the DOS prompt:

```
C> WIN PRO386W /V1
```

You can also configure the banner by setting the /V command line switch on the command line property of Prologs program item.

/Z - The 3D-Look Dialog Switch

Another command line switch lets you can decide whether or not the environment uses 3d-look dialogs. The 3d-look dialogs make use of the CTL3DV2.DLL file which should be located in your Windows directory. If this file is not present all dialogs will appear as normal. The /Z switch can take either of the following values:

Switch	**Meaning**
/Z0	use 3d-look dialogs
/Z1	use normal dialogs

For example, you can set Prolog to start up without 3d-look dialogs by entering the following at the DOS prompt:

```
C> WIN PRO386W /Z1
```

You can also configure the look of the system dialogs by setting the /Z command line switch on the command line property of Prologs program item.

Errors and Error Numbers

The Prolog error numbers are organised in logical groups. The errors and their messages with their logical groups are listed below:

Configurable Memory Errors

1 Backtrack Stack Full
2 Local Stack Full
3 Reset Stack Full
4 Heap Space Full
5 Text Space Full
6 Program Space Full
7 System Stack Full
8 Input Space Full
9 Output Space Full

Console I/O Errors

10 Window Handling Error
11 Keyboard Break
12 Mouse Handling Error
13 Graphics Handling Error
14 Console Buffer Full

Predicate and Control Errors

20 Predicate Not Defined
21 Control Error
22 Instantiation Error
23 Type Error
24 Domain Error
25 Too Many Arguments
26 Term Too Big

File Handling Errors

30 File Handling Error
31 File Not Found
32 Path Not Found
33 Too Many Files Open
34 File Access Denied
35 Disk Full
36 Memory Full

Term I/O Errors

40 Format Not Defined
41 Format Field Overflow
42 Syntax Error
43 End Of File
44 Binary Format Error
45 Checksum Error
46 String Too Long
47 Atom Too Long

Arithmetic Handling Errors

50 Function Not Defined
51 Arithmetic Underflow
52 Arithmetic Overflow
53 Arithmetic Error

Assembly and Compilation Errors

60 Instruction Not Defined
61 Bad Number Of Arguments
62 Bad Argument Type
63 Bad Register Number
64 Label Not Defined
65 Label Already Defined
66 Bad Assembler Module
67 Predicate Protected

DOS Errors

10xx Unknown Error

Memory Errors

Memory errors occur when the execution of a program causes one of Prolog's internal memory areas to become full. The general solution to these errors, is to halt and then run Prolog with the appropriate memory area increased. You can configure the memory areas by setting the command line switches on the command line property of Prologs program item.

Error 1 Backtrack Stack Full

This error message is displayed when the internal backtrack stack is full. The program that was running, when the error occurred, had too many choice points in the current evaluation. To solve this you should halt from the current session and try running Prolog with a larger backtrack stack. For example, you can set the backtrack stack of Prolog to have a size of 64k by entering the following at the DOS comand line:

```
C> WIN PRO386W /B64
```

You can also configure the backtrack stack by setting the /b command line switch on the command line property of Prologs program item.

Error 2 Local Stack Full

This error message is displayed when the internal local stack is full. The program that was running, when the error occurred, had too great a depth of non-tail recursive calls. To solve this you should halt from the current session and try running Prolog with a larger local stack. For example, you can set the local stack of Prolog to have a size of 64k by entering the following at the DOS comand line:

```
C> WIN PRO386W /L64
```

You can also configure the local stack by setting the /l command line switch on the command line property of Prologs program item.

Error 3 Reset Stack Full

This error message is displayed when the internal reset stack is full. The evaluation that was current, when the error occurred, contained too many variables that might need unbinding. To solve this you should halt from the current session and try running Prolog with a larger reset stack. For example, you can set the reset stack of Prolog to have a size of 64k by entering the following at the DOS comand line:

```
C> WIN PRO386W /R64
```

You can also configure the reset stack by setting the /r command line switch on the command line property of Prologs program item.

Error 4 Heap Space Full

This error message is displayed when the internal heap space is full. The program that was running, when the error occurred, had created too many structures or too large a structure. To solve this you should halt from the current session and try running Prolog with a larger heap space. For example, you can set the heap space of Prolog to have a size of 256k by entering the following at the DOS comand line:

```
C> WIN PRO386W /H256
```

You can also configure the heap space by setting the /h command line switch on the command line property of Prologs program item.

Error 5 Text Space Full

This error message is displayed when the internal text space is full. The program that was running, when the error occurred, had created too many atoms or strings. To solve this you should halt from the current session and try running Prolog with a larger text space. For example, you can set the text space of Prolog to have a size of 512k by entering the following at the DOS comand line:

```
C> WIN PRO386W /T512
```

You can also configure the text space by setting the /t command line switch on the command line property of Prologs program item.

Error 6 Program Space Full

This error message is displayed when the internal program space is full. This type of error, indicates that there are too many programs currently in memory. It normally occurs during a consultation or assertion. To solve this you should halt from the current session and try running Prolog with a larger program space. For example, you can set the program space of Prolog to have a size of 4096k by entering the following at the DOS comand line:

```
C> WIN PRO386W /P4096
```

You can also configure the program space by setting the /p command line switch on the command line property of Prologs program item.

Error 7 System Stack Full

This error message is displayed when the system stack is full. The program that was running tried to process a structure that was too deep. To solve this you should halt from the current session and try running Prolog with a larger system stack. For example, you can set the system stack of Prolog to have a size of 256k by entering the following at the DOS comand line:

```
C> WIN PRO386W /S256
```

You can also configure the system stack by setting the /S command line switch on the command line property of Prologs program item.

Error 8 Input Space Full

This error message is displayed when the input string buffer is full. The program that was running tried to set input to a string that was too big. To solve this you should halt from the current session and try running Prolog with a larger string input buffer. For example, you can set the string input buffer of Prolog to have a size of 1Mb by entering the following at the DOS prompt:

```
C> WIN PRO386W /I1024
```

You can also configure the string input buffer by setting the /I command line switch on the command line property of Prologs program item.

Error 9 Output Space Full

This error message is displayed when the output string buffer is full. The program that was running tried to output too many characters to a string. To solve this you should halt from the current session and try running Prolog with a larger string output buffer. For example, you can set the string output buffer of Prolog to have a size of 1Mb by entering the following at the DOS prompt:

```
C> WIN PRO386W /O1024
```

You can also configure the string output buffer by setting the /O command line switch on the command line property of Prologs program item.

Console I/O Errors

The console I/O errors deal with the interface between user programs and the console. These errors include: incorrect window handling, keyboard interruption of running programs and attempts at mouse and graphics handling when the appropriate drivers have not been installed.

Error 10 Window Handling Error

This error occurs when an attempt at a window operation is made in the wrong mode or on a window that doesn't exist.

Error 11 Keyboard Break

This error occurs when the ctrl-<Break> or ctrl-C keys are pressed. The keyboard break error can be used to break into Prolog programs while they are running.

Error 12 Mouse Handling Error

This error occurs when an attempt to access the mouse is made and the mouse driver is not loaded. To rectify this problem halt from Prolog and load the mouse driver.

Error 13 Graphics Handling Error

This error occurs when an attempt is made at a DOS graphics operation and the graphics driver GRAFIX.EXE is not loaded. To rectify this problem halt from Prolog and load the graphics driver.

Error 14 Console Buffer Full

This error message is displayed when the console buffer becomes full. This type of error will occur if you attempt to enter a single term at the user device that exceeds the console buffer space. This does not affect terms read from any other file device or window.

Predicate and Control Errors

The predicate and control errors deal with the calling of undefined predicates or the calling of predicates with incorrect arguments.

Error 20 Predicate Not Defined

This error occurs when you try and call a non-dynamic predicate that is not currently defined. This error can be avoided if the predicate is declared dynamic, using *dynamic/1*, in which case the call to the undefined predicate will simply fail.

Error 21 Control Error

This error occurs when there is an invalid call structure. It will occur when the call being executed is not of the correct form (e.g. when a meta-variable is unbound or has been bound to a structure that is not an atom).

Error 22 Instantiation Error

This error occurs when a built-in predicate expects a bound argument and receives a variable instead.

For example the following call generates an instantiation error:

```
?- fcreate(X,Y).
```

because *fcreate/2* expects both of its arguments to be bound.

Error 23 Type Error

This error occurs when a built-in predicate expects a specific type of argument and receives the wrong type.

For example the following call generates a type error:

```
?- beep(23,fred).
```

because *beep/2* expects an integer as its second argument.

Error 24 Domain Error

This error occurs when a built-in predicate expects an argument to be in a specific range and receives data that is outside that range.

Error 25 Too Many Arguments

This error occurs when you try and call a predicate with more than 128 arguments.

Error 26 Term Too Big

This error occurs when you try to create a term that exceeds the limits for term size.

File and Memory Handling Errors

The file and memory handling errors deal with the situations where file and disk and external memory accessing is invalidated.

Error 30 File Handling Error

This error is the generic file handling error, it occurs when an attempt is made to access a file that is not currently open to Prolog. This error can be avoided by opening the file using the *fopen/2* predicate.

Error 31 File Not Found

This error occurs when an attempt is made to open, delete, rename or execute a file that is not on the current disk (or on the path specified).

Error 32 Path Not Found

This error occurs when an attempt is made to open, delete, rename or execute a file and the path element for that file does not exist.

Error 33 Too Many Files Open

Prolog only allows 8 files to be open at any one time. If you have exceeded his limit error 33 will be generated. To alleviate this problem close one or more files using *fclose/1* before opening the next.

Error 34 File Access Denied

This error occurs when you try to write to a file that has been opened in read only access mode or try to read from a file that has been opened in write only access mode. To avoid this error open the file in read/write mode using *fopen/2*.

Error 35 Disk Full

This error indicates that your hard disk is full and usually occurs when you are writing to the disk or when you are copying files from within Prolog.

Error 36 Memory Full

This error is displayed when the external memory is full This may occur if there are a number if memory resident programs installed or too many Prolog windows have been created. One way to alleviate this problem is to halt from the current session and try quitting from some of the memory resident programs. Alternatively you could set Prolog to use less external memory by configuring Prolog's memory areas to use less memory.

Term I/O Errors

Error 40 Format Not Defined

This error occurs when you try to call either of the predicates *fwrite/4* or *fread/4* with an undefined format.

Error 41 Format Field Overflow

This error occurs when performing input or output using *fwrite/4* or *fread/4* if the term to be input or output is larger than the specified fieldwidth. To solve this problem you can either use a free field width (a field width of 0) or on output use truncated output.

Error 42 Syntax Error

This error occurs when a Prolog term read in with *read/1* contains a syntax error. You can set four different behaviours for the *read/1* predicate when it encounters a syntax error. This can be done by setting the "syntax_errors" prolog flag.

Error 43 End Of File

This error occurs when you read past the end of a file.

Error 44 Binary Format Error

This error occurs when a Prolog object code file is being loaded and has been corrupted in some manner.

Error 45 Checksum Error

This error occurs if there is corruption in a compressed data record. Such corruption is discovered on completion of decompressing the record, when the record's checksum is tested.

Error 46 String Too Long

This error occurs when an attempt is made to create a string that exceeds the string buffer. To solve this you should halt from the current session and try running Prolog with a larger string input or output buffer size, depending whether the error occurred during string input or output. For example, you can set both the string input and output buffer sizes of Prolog to have a size of 1Mb by entering the following at the DOS comand line:

```
C> WIN PRO386W /I1000 /O1000
```

You can also configure the string input and output buffers by setting the /I and /O command line switches on the command line property of Prologs program item.

Error 47 Atom Too Long

This error occurs when an attempt is made to create an atom that exceeds 1024 characters. This problem can be avoided by splitting the atom into smaller parts.

Arithmetic Handling Errors

The errors associated with the following predicates:

is/2	</2	=:=/2	=</2
=\=/2	>/2	>=/2	seed/1

Error 50 Function Not Defined

This error occurs when the arithmetic predicate *is/2* is used with an undefined function.

Error 51 Arithmetic Underflow

This error occurs when the value of a number is less than the smallest number that can be represented. In LPA Prolog this occurs for numbers whose absolute value is less than 2.2e-308.

Error 52 Arithmetic Overflow

This error occurs when the value of a number is greater than the largest number that can be represented. In LPA Prolog this occurs for numbers whose absolute value is greater than 1.7e308.

Error 53 Arithmetic Error

This error occurs when the expression handler tries to perform an invalid operation such as finding the square root of a negative number.

Assembly and Compilation Errors

Some of the following errors should not occur during the normal running of Prolog. If the indicated errors are reported please contact LPA directly.

Error 60 Instruction Not Defined

The optimising compiler has generated a Prolog machine instruction that is not part of the internal language. This is a bug which should never occur in the release system.

Error 61 Bad Number Of Arguments

The optimising compiler has generated a Prolog machine instruction with the wrong number of arguments. This is a bug which should never occur in the release system.

Error 62 Bad Argument Type

The optimising compiler has generated a Prolog machine instruction with an argument of the wrong type. This is a bug which should never occur in the release system.

Error 63 Bad Register Number

The optimising compiler has generated a Prolog machine instruction with an invalid register number. This is a bug which should never occur in the release system.

Error 64 Label Not Defined

The optimising compiler has generated a Prolog machine instruction which attemts to reference an undefined code label. This is a bug which should never occur in the release system.

Error 65 Label Already Defined

The optimising compiler has generated a given code label more than once in a given relation. This is a bug which should never occur in the release system.

Error 66 - Bad Assembler Module

A user-defined assembler code module has an invalid format.

Error 67 Predicate Protected

An attempt has been made to abolish or otherwise modify a locked relation, or to assert to or retract from an optimised relation. Normally, you should not attempt to modify such relations.

DOS Errors

A range of errors reflecting directly the MS-DOS operating system errors.

Error 10xx Unknown Error

Errors that occur above 1000 are MS-DOS errors. The actual error can be found by subtracting 1000 from the error number and referring to your MS-DOS documentation.

ASCII, ANSI and OEM Character Sets

One of the difficulties in maintaining compatibility between the Windows and DOS versions of Prolog is that the character set used by Windows is different from that used by DOS. Fortunately, the lower 128 characters of both sets conform to the 7-bit ASCII standard, which contains all of the English upper and lower case letters, numerals, punctuation symbols and brackets. Windows and DOS diverge, however, in their treatments of the upper half of the table, namely characters 128-255.

All accented letters, as well as the majority of currency symbols and graphics characters, are stored in the upper half of the character table. The problem is that the IBM PC set, as used by DOS, is organised differently from the ANSI set used by Windows. For example, the UK "Pound" sign ("£") is IBM PC code 156, while under Windows it is 163.

Atom - Data Type

Atoms are text names that are used to identify data, programs, modules, files, windows, and so on.

The maximum length of an atom is 1024 characters. There are four types of atoms: alphanumeric, symbolic, quoted and special atoms.

Alphanumeric Atoms

An alphanumeric atom is one that starts with a lower-case letter (a-z) and is only followed by: alphabetic characters (A-Z,a-z), extended ASCII alphabetic characters (ASCII codes 128-154 and 160-167), digits (0,9) and underscores.

The following are all alphanumeric atoms:

apple a1 apple_cart longTable

Symbolic Atoms

A symbolic atom is written as a sequence of symbolic characters, and characters in the upper half of the 8-bit ASCII table. The symbolic characters are:

#	$	&	=	-	^	~	\	@
`	:	.	/	+	*	?	<	>

The following are all symbolic atoms:

&	&:	++	<<	>>	<--	..	*-/*

Note that the /* appearing in the last example is not interpreted as the start of a comment.

Quoted Atoms

A quoted atom is any sequence of characters surrounded by single quotes. To insert a single quote character in a quoted atom use two adjacent single quote characters:

```
''
```

The tilde character (~) is used within quoted atoms as an escape character. Tilde followed by a printable character in the range '@' to '_' is used to represent a control character. For example:

```
'~I'
```

represents ctrl-I.

The tilde character can also be followed by a three digit number representing the ASCII code of a character. This can be useful for inserting characters with an ASCII value greater than 127.

To insert a tilde in a quoted atom use ~~. The following are all quoted atoms:

```
'Apple'    '123'    'hello world'    '~Ibold~M~J'    '~065'    'don''t care'
```

The last example represents the atom:

```
don't care
```

Special Atoms

The special atoms are as follows:

! ; [] {}

Backtracking

The process of re-evaluating a Prolog goal to try to find more solutions.

Byte List - Data Type

A byte list (also referred to as <char_list>) is a sequence of characters surrounded by the double quotes character ("). It is simply an abbreviation for the list of decimal integer ASCII codes of the characters in the sequence. For example, the byte list:

```
"A boy"
```

is simply a shorthand form of the list:

```
[65,32,98,111,121]
```

To insert a double quote character in a byte list use two adjacent double quote characters:

```
""
```

As with quoted atoms the tilde character is used an escape character, allowing you to enter control characters in a byte list. For example:

```
"~G"
```

represents the list:

```
[7]
```

(To insert a tilde in a byte list use ~~.)

Character Set

LPA Prolog uses the full 8-bit ASCII character set, although all characters with special meaning are confined to the first 128 characters (the 7-bit ASCII set). The upper half of the character table usually contains international characters, accents, and graphics characters, and it tends to be machine specific.

Character Sets, Typefaces and Fonts

Windows has two character sets, OEM and ANSI: these differ only in the logical assignment of characters to binary codes, and do not imply anything about the appearance of text.

Windows supports many typefaces, and most can be scaled and styled to produce a myriad of different fonts. The font used to display text is of no concern to Prolog, just so long as its character set (logical assignment) is compatible with Prolog syntax.

There is only one standard Windows typeface which supports the OEM character set, and because this is non-scaleable and therefore of fixed size and style, it is also a font: for convenience, this is referred to as the "OEM font". Windows uses another typeface, again non-scaleable, as its default for dialogs and controls: we call this one the "ANSI font" (of course, nearly all typefaces and their derivative fonts use the ANSI character set).

Because the OEM and ANSI fonts (sic) are so central to both Prolog and Windows, they are predeclared and given special numeric font names:

Whenever the font of a window is set using *wfont/2*, and also in the cases of certain built-in dialog predicates (such as *abtbox/3* and *sttbox/2*, either of these two fonts can be used directly.

Clauses

Clauses are the building blocks of Prolog programs. There are two types of clause: facts and rules.

A fact is of the form:

head.

where *head* is the head of the clause. *head* may be an atom or a compound term whose functor is any atom except :-. The fact is terminated by a '.' followed by a white space character (e.g. a space, or a carriage return).

A rule is of the form:

head :- *t1*, *t2*, …, *tk*. ($k \geq 1$)

where *head* is the head of the clause and the terms to the right of :- are the body of the clause. Each *tk* is known as a call term or goal. A call term must be an atom (a 0-argument call), a compound term, or a variable name. The rule is terminated by a '.' followed by a white space character.

The functor of the head of a clause is the predicate that the clause describes. All the clauses describing a given predicate comprise its definition. The arity of a clause is the number of arguments in its head.

```
foo.                                    % a fact
foo(1) :- bar.                          % a rule

likes(Anyone, prolog) :-                % a rule
    logic_programmer(Anyone).

likes(Anyone, Anything).                % a fact

my_append([], X, X).                    % a fact
my_append([A|B], C, [A|D]) :-           % a rule
    my_append(B, C, D).

is_not_true(X) :-
    X,
    !,
    fail.
is_not_true(X).
```

These clauses define the relations foo/0, foo/1, likes/2, my_append/3 and is_not_true/1 respectively.

All clauses describing a predicate must be in a single source file unless the predicate is declared as multifile (see multifile/1).

Comments

Comments have no effect on the behaviour of a program. In fact they are ignored when a Prolog term or program is read in. There are two forms of comment:

1. A sequence of characters that begins with the symbol /* and ends with */ is treated as a comment.

2. A sequence of characters that begins with the symbol % and ends with the end-of-line character (carriage return) is treated as a comment.

The first type of comment allows a comment to extend over several lines. The second type of comment is useful when commenting a single line. For example:

```
/*
** this is a comment
*/
% so is this
```

Compound Term - Data Type

A compound term is a structured data item that consists of a functor followed by a sequence of one or more arguments which are enclosed in brackets and separated by commas. The general form of a compound term is:

functor(*t1*, *t2*, …, *tn*) $n \geq 1$

functor is the functor. It can be an atom or a variable name. (For further details about the use of a variable name as the functor please see the section below entitled "Meta-variables").

The term *ti* represents the *i*'th argument of the compound term.

The arity of a compound term is the number of arguments it has (*n* in the example above). We refer to functor with arity *n* using the notation:

functor/*n*

The following are examples of compound terms:

```
likes(paul,prolog)          % functor is likes (arity is 2)
read(X)                     % functor is read (arity is 1)
>(3,2)                      % functor is > (arity is 2)
```

A compound term can be thought of as representing a record structure. The functor represents the name of the record, while the arguments represent the record fields.

Note: There must be no space between the functor and the opening parenthesis of a compound term. For example:

```
likes   (paul,prolog)
```

is not a legal compound term. Spaces between the arguments are allowed however.

Data Types

The following data types are used in LPA Prolog.

atoms

byte lists

compound terms

floating point numbers

integers

lists

strings

variables

Directives

A directive is a Prolog term of the form:

```
:- goal1, ..., goalk.                    % k ≥ 1
```

where *goali* is a call term (i.e. goal). A command is executed automatically when it is encountered during a consult or reconsult (see also *initialization/1*).

```
:- write('hello world'),nl.
```

Floating Point Number - Data Type

A floating point number is written as an *optional* minus sign (-) followed by a sequence of one or more digits followed by a decimal point (.) followed by one or more digits, *optionally* followed by an exponent. An exponent is written as e (or E) followed by an optional minus sign followed by one to three digits. Note that in LPA Prolog floating point numbers are 64-bit values in the range [2.2e-308..1.7e308].

As with integers, the plus sign (+) must not be used to denote a positive floating point number. For example:

```
1.0     246.8091     -12.3     20.003e-10     -1.3E102
```

The following are *not* floating point numbers:

```
.9                  % does not start with a digit
3e-22               % no decimal point
34.1 e3             % contains a space before the 'e'
-.7                 % no digit after the minus sign
56.1e4.8            % exponent is not an integer
23.                 % no digit after the decimal point
```

Grammar Rules

A Prolog program may contain one or more grammar rules. These grammar rules may be used to define the syntax of a language and to define a parser for that language.

A grammar rule takes the form:

grammar_head --> grammar_body.

Where *grammar_head* is a non-terminal symbol optionally followed by a terminal symbol. The body of the grammar rule is a sequence of terminals, non-terminals or grammar conditions, each separated by commas or semi-colons. A grammar condition is a sequence of Prolog call terms surrounded by curly brackets ('{' and '}').

For a detailed description of the Prolog grammar rules please see the chapter on 'Grammar Rules'.

```
sentence --> noun_ph, verb_ph.
verb_ph --> verb, noun_ph.
verb --> [likes] ; [hates].
noun_ph --> determiner, noun.
determiner --> [the].
noun --> [boy] ; [dog].
```

IBM ASCII Character Set

An 8-bit character set, which using only the first seven bits (values 0-127 inclusive), sets a standard code for the English upper and lower case letters, numerals, punctuation symbols and brackets.

The IBM extended ASCII character set is defined by the upper half of the 8-bit ASCII character set (values 128-255 inclusive) contains non-English alphabetic characters and punctuation, as well as some block graphics characters.

Integer - Data Type

An integer is a number with no fractional part. It is written as a sequence of digits, optionally preceded by a minus sign (-). Note that in LPA Prolog integers are 32-bit values which gives them a range of [-2147483648..2147483647].

The plus sign (+) must not be used to denote a positive integer. All positive integers are written without a leading sign character. For example:

```
0       1       9821       -10       -64000
```

List - Data Type

A list is a sequence of terms of the form:

[*t1*,*t2*,…,*tn*] $n \geq 0$

The term *ti* is the *i*'th element of the list. It can be any type of Prolog term. The simplest form of list is the empty list (*n* = *0*):

```
[]
```

The following example is a four element list:

```
[[a,list,of,lists],and,numbers,[1,2,3]]
```

Unknown elements of a list can be represented by variables. For example:

```
[X,Y,Z]
```

We also represent a list using the notation:

[*t1*, *t2*, …, *ti* | *Variable*] $i \geq 1$

This list pattern represents a list that begins with the terms *t1*,*t2*,…,*ti* with the remainder of the list (the tail) denoted by *Variable*.

For example the list pattern:

```
[Head|Tail]
```

could be unified with the list:

```
[1,2,3,4]
```

to give the variable bindings:

```
Head = 1
Tail = [2,3,4]
```

Meta-variables

A meta-variable is a variable which appears in place of a callable Prolog structure. LPA provides full support for the usual Edinburgh predicates *=../2* and *call/1*, but the following methods for supporting meta-calling are far more efficient.

Condition meta-variables can be bound to atoms and compound terms and called directly without using *call/1*.

Predicate meta-variables can be used where the functor of a compound term is a variable that is bound at the time of calling the term.

Condition Meta-variable

This is where a variable appears as a goal in the body of a rule. The head of a clause may not be represented in this way. By the time the meta-variable is called it must have been instantiated to one of the following.

An atom (represents a call to a 0-argument relation).

A compound term of the form:

relation(*t1*, *t2*, …, *tk*)

The effect of evaluating a condition meta-variable is the same as if the condition had appeared in the source program instead of the meta-variable.

The following is the definition of the built-in predicate *\+/1*:

```
\+(X) :- X, !, fail.
\+(X).
```

You can query this predicate as follows:

```
\+(true).
no

\+(false)
yes

\+(compare(=, 2, 3)).
yes
```

Predicate Meta-variable

This is where a goal in the body of a rule is a compound term whose functor is a variable. By the time the goal is evaluated, the meta-variable must have been bound to an atom.

```
map(Pred, [], []).
map(Pred, [X|Y], [X1|Y1]) :-
    Pred(X, X1),
    map(Pred, Y, Y1).
```

In this example, it is assumed that the meta-variable 'Pred' will be bound to the name of a binary relation. Given the following binary relations:

```
double(X, Y) :-
    Y is X + X.

square(X, Y) :-
    Y is X * X.
```

You can query map/3 as follows:

```
map(double, [1,2,3,4], X).
X = [2,4,6,8]

map(square, [1,2,3,4], X).
X = [1,4,9,16]
```

Note: In standard Edinburgh syntax, the call to Pred(X,X1) in the second clause for map/3 would have to be replaced with calls to *=../2* and *call/1* as follows:

```
map(Pred, [X|Y], [X1|Y1]) :-
    Call =.. [Pred,X,X1],
    call(Call)
    map(Pred, Y, Y1).
```

Modifying Dynamic Code During Its Execution

There are two types of dynamic code 'normal' and 'logical' (see *dynamic*/1). It is potentially dangerous to modify 'normal' dynamic code that is still being executed, and every care should be taken not to do so. If you are running into this problem you can make all the dynamic code in the system *safely* 'logical' using the /D command line switch when you boot up Prolog. Using 'logical' dynamic predicates is completely safe in this respect although the code runs at a slightly slower speed.

Number Bases

You can enter numbers in a particular base using the notation. As in the following examples:

```
?- X = 16'F.                          % a number in base 16
X = 15

?- ?- X = 2'10010101.                 % a number in base 2
X = 149
```

Incidentally you can also use this notation to give the ASCII code of the character following the quote sign, as in the following:

```
?- X = 0'a.
X = 97
```

Operators

Operators allow you to use an alternative syntax for compound terms. There are three types of operator: prefix, postfix and infix. Each particular operator type may have a different precedence and associativity.

Prefix Operators

The compound term:

functor(*term*)

can also be written as:

functor term

if *functor* has been declared a prefix operator, using op/3. For example, the built-in predicate *spy/1* is a prefix operator which means that the following compound term can be entered:

```
spy my_module
```

This is equivalent to:

```
spy(my_module)
```

Postfix Operators

The compound term:

functor(*term*)

can also be written as:

term functor

if *functor* has been declared a postfix operator, using op/3. For example, if is_male/1 has been declared a postfix operator then you could enter the compound term:

paul is_male

This is equivalent to:

is_male(paul)

Infix Operators

The compound term:

functor(*term1,term2*)

can also be written as:

term1 functor term2

if *functor* has been declared an infix operator, using op/3. For example, the built-in arithmetic function '+' is an infix operator which means you can enter the compound term:

```
5 + 10
```

This is equivalent to the predicate:

```
+(5,10)
```

Note: there is no definition for the +/2 predicate (+ is simply an argument given to the is/2 predicate). so entering this at the command line will give:

```
| ?- 5 + 10 .
! ---------------------
! Error 20: Predicate Not Defined
! Goal       : 5 + 10
```

Operator Precedence

The preference given to an operator in the reading of a term. The lower the precedence, the more strongly an operator binds to its arguments.

For example, the expression:

```
2 + 5 * 8
```

represents the term:

```
+(2,*(5,8))
```

because * has a lower precedence than +.

Operator Associativity

The associativity of an operator clears any ambiguity in an expression that contains two operators of the precedence.

Operator Types

The type of an operator defines its associativity. It is used to disambiguate an expression that contains two operators of the same precedence. If an operator is non-associative then its arguments must be sub-expressions of strictly *lower* precedence than the operator itself.

A left associative operator is one whose left hand argument may be a sub-expression of the same precedence as the operator itself (it can also be lower). A right associative operator is one whose right hand argument may of the same (or lower) precedence as the operator. For example, the built-in operators '+' and '-' are both left associative infix operators with a precedence of 500. This means that the expression:

```
10-5+2
```

represents the compound term:

```
+(-(10,5),2)
```

because the left hand argument of '+' can have the same precedence. The '-' operator *cannot* have a right argument with the same precedence. This means that the following compound term is *not* a valid interpretation of the above expression:

```
-(10,+(5,2))
```

because the right hand argument would have the same precedence as '-' itself (and '-' is not right associative).

Note that these types indicate the associativity and position of an operator.

Declaring Operators

Operators are declared using the built-in predicate *op/3*. The form of this predicate is:

```
op(+Precedence, +Type, +Name)
```

where *Precedence* is the operator's precedence (an integer in the range 1 to 1200), *Type* defines the operator type and associativity (e.g. fx), and *Name* is the name of the operator (or a list of operator names). If *Precedence* is 0 then the operator declaration for *Name* is cancelled.

The following examples show how some of the built-in operators are defined.

```
op(200, xfy, ^).
op(500, fx, [+, -]).
```

It is possible to have more than one operator of the same name. For example, the built-in operator '+' is declared as both a prefix and an infix operator. The built-in predicate *current_op/3* can be used to find out what operators are currently defined. The format of this predicate is:

```
current_op(?Precedence, ?Type, ?Name)
```

This succeeds if there is an operator called *Name* of type *Type* and with a precedence of *Precedence*. It can be used to backtrack through the list of currently defined operators.

Predicate Definitions

When predicate definitions are given, the functor, arguments and positions of the arguments of the predicate are shown as a template such as:

foo(*+Arg1*, *?Arg2*, *-Arg3*)

This defines a predicate called foo that can take three arguments. The character that precedes each argument name is a mode declaration.

Program Structure

A Prolog program is made up of a selection of the following program elements:

comments

clauses

grammar rules

directives

References

K.L.Clark and F.G.McCabe. *micro-PROLOG: Programming in Logic*. Prentice-Hall International, 1984. (This book does not describe the Edinburgh syntax.)

W.F.Clocksin and C.S.Mellish. *Programming in Prolog*. Springer Verlag, 1987.

I.Bratko. *Prolog Programming For Artificial Intelligence*. Addison-Wesley Publishing Company, 1986.

R.A.Kowalski. *Logic For Problem Solving*. Artificial Intelligence series. North Holland Inc.

T.Dodd. *Prolog: A Logical Approach.* Oxford University Press

Logic Programming Associates Ltd do not endorse these books or recommend them over others on the same subject.

Separators and Terminators

The normal term separator is the comma. This is used to separate terms in lists and argument lists. A space must be used to separate an operator from an operand if they are both of the same token type (e.g. they are both alphanumeric tokens).

The usual term terminator is the full stop followed by a space or carriage return. (A full stop not followed by a space or carriage return is treated as a symbolic atom - see below.)

Note that a space between an atom and a left parenthesis, '(', is significant.

String - Data Type

A text data type specific to LPA Prolog. This data type is capable of storing any length of text. The limit on the size of strings is set separately for input and output using the /I and /O command-line switches.

Use backwards quotes to define a string:

```
`This is a string`
```

To insert a backwards quote character in a string use two adjacent backwards quote characters:

```
`I don``t care`
```

Terms

Terms are the fundamental data types in Prolog. They are the building blocks from which Prolog clauses, and commands are constructed.

The basic term types are: variables, integers, floating point numbers, atoms, strings (a unique LPA text data type), lists, byte lists (normally called strings in 'Edinburgh' parlance) and compound terms.

Unification

The process of matching two terms during the running of a Prolog program. Terms are said to match under the following conditions:

- Atomic terms match if they are identical.
- A variable will match with any atomic or compound term.
- Two unbound variables will match and become the same variable.
- Two compound terms will match if they have the same functor and each of the arguments in one term can be matched with its corresponding argument in the other term.

The last type of unification takes place during the evaluation of a query, where an attempt is made to match the goal with the head of a clause in the database.

Variable - Data Type

A variable name is an alphanumeric sequence of characters beginning with an upper case letter (A-Z) or an underscore ('_'). The alphanumeric sequence can include '_' and characters from the upper half of the ASCII table. For example, the following are variable names:

```
Anything    _var    _1    X    Var1
```

Quoting with single quotes overrides the variable name convention. For example the following are both quoted atoms:

```
'Anything'    '_var'
```

An underscore on its own is an anonymous variable.

<number>

A number is either an integer or floating point number.

<char>

An integer in the range [0..255] that is a character code.

For example, the character code for the letter 'a' is 97.

<char_list>

A list of integers each of which is in the range [0..255].

For example the following list represents the characters 'a', 'b' and 'c'.

```
[97,98,99]
```

This list can also be written as:

```
"abc"
```

Lists of this kind are also known as byte lists.

<expr>

An arithmetic expression or number. For more information on arithmetic expressions and the functions allowed see is/2.

<integer_expr>

An integer or an expression which evaluates to an integer.

For example, the following are all integer expressions:

```
1          1 + 2          10 // 1
```

<functor>

An atom that denotes the functor of a predicate. This atom cannot be the empty list [].

For example given the following predicate:

```
foo(a,b).
```

The functor of this predicate is foo. and the predicate specification is foo/2.

<arity>

An integer that denotes the number of arguments to a predicate. In the current system this integer is in the range [0..64].

For example given the following predicate:

```
foo(a,b,c).
```

The arity of this predicate is 3. and the predicate specification is foo/3.

<pred_spec>

A predicate specification is of the form:

Functor/*Arity*

where *Functor* is an atom that is the functor of the predicate and *Arity* is an integer that specifies the arity, or number of arguments of the predicate.

For example given the following predicate:

```
foo(X) :-
    write(hello).
```

its predicate specification would be:

```
foo/1
```

denoting a predicate with one argument and whose functor is foo.

<pred_specs>

A list of predicate specifications of the form functor/arity.

For example:

```
[foo/1,foo/2,bar/4]
```

This denotes three distinct predicates.

<file_spec>

A file specification may be either an atom or a compound term.

If the specification is an atom it refers to the file name and may or may not include a path and extension.

If the specification is a compound term, then it should be of the form:

PathAlias(*FileName*).

Where *PathAlias* refers to a "logical" path alias set up using the file_search_path/2 database and *FileName* is an atom that denotes a file that may or may not include a path and extension. The resultant file is found by taking the *FileName* relative to the directory given by the *PathAlias*.

For example the following file specification refers to the file FOO.PL in the Prolog EXAMPLES directory:

```
examples('FOO.PL')
```

<file_name>

A file name is an atom that refers to a particular file by name and may or may not include a path and extension.

For example, the following atom refers to the file FOO.PL located in the absolute directory C:\MYFOO:

```
'C:\MYFOO\FOO.PL'
```

The following atom refers to the file FOO.PL located in the current directory:

```
'FOO.PL'
```

The following atom may refer to the file FOO without an extension, but a default extension may be given to the file by the predicate that is using the file specification:

```
'FOO'
```

<path_alias>

A path alias is an atom that refers to a "logical" path name set up using the file_search_path/2 database. It is used as the functor of a file specification to specify the location of a file.

For example if the following file_search_path/2 clause has been asserted:

```
file_search_path(myfiles,'C:\MYFILES')
```

then the following file specification refers to the file FOO.PL in the C:\MYFILES directory:

```
myfiles('FOO.PL').
```

<file_specs>

A list of file specifications.

For example the following list of file specifications refers to three different files:

```
['C:\PROLOG\FOO.PL',bar,myfiles('FRED.PL')]
```

<list_of Type>

A list consisting of terms of the given type.

For example, the following list of atoms is of the type <list_of <atom> >:

```
[a,b,c,d,e]
```

<conjunct_of Type>

A conjunction of terms of the given type.

For example the following conjunction of strings is of the type <conjunct_of <string> >

```
(`a`,`b`)
```

<goal>

A goal is a compound term or an atom that represents a call to a Prolog program.

For example, a goal term is:

```
write(hello)
```

If a variable is instantiated to a goal and occurs as a call it is known as a meta-variable, as in the following program:

```
foo(Goal) :-
      Goal.
```

The following instantiates the *Goal* in the above example to the goal term write(hello), which is then executed:

```
?- foo(write(hello)).
hello
```

<window_handle>

There are four types of window handle. These are:

An atom that refers to a named top-level window created by Prolog.

A conjunction of an atom, that refers to a named top-level window created by Prolog, and an integer that refers to the ID of a control item within the window.

An integer that refers to the handle of any window (not just those created by Prolog). Handles are assigned by Windows itself and may be found using wfind/3, wlink/3 and wndhdl/2.

A conjunction of an integer, that refers to the handle of any window, and another integer that refers to the ID of a child of the window.

,/2

conjunction

Call1 , *Call2*

+Call1 <goal>

+Call2 <goal>

Succeeds if both *Call1* and *Call2* succeed.

A call meta-variable can be bound to a conjunction of goals.

See Also
;/2

!/0

control backtracking

The predicate *!/0* always succeeds, but it has two side effects which are used to control backtracking.

See Also
one/1

->/2

if then control predicate

If >/2 Then

+If	<goal>
+Then	<goal>

The predicate *>/2* acts as a local cut. It prevents backtracking into *If*. Note that commas bind more strongly than the '>' operator because the operator precedence of comma is 1000 while the operator precedence of > is 1050.

See Also

:/2

;/2

disjunction

Either ; *Or*

+Either	<goal>
+Or	<goal>

Succeeds if *Either* succeeds or *Or* succeeds. A call meta-variable can also be bound to a disjunction of goals.

See Also

./2
->/2

</2

check that one expression is less than another

E1 < *E2*

+E1	<expr>
+E2	<expr>

Succeeds if the result of the arithmetic expression *E1* is less than the value of the arithmetic expression *E2*.

See Also

=</2
>/2
>=/2
=\=/2
=:=
is/2

<~/2

re-direct input to a file or a string

Call <~ *In*

?In	<file_spec> or <string>
+Call	<goal>

If *In* is a file specification, all input performed by the *Call* is re-directed from that file. If the *Call* is re-satisfiable, then on backtracking into the *Call* any further input is also re-directed from that file. Note: the file is not closed automatically.

If *In* is a string, all input performed by the *Call* is re-directed from that string. Note: in this form <~/2 is not re-satisfiable even if the *Call* is.

See Also
~>/2

=../2

defines the relationship between a structure/atom and a list

Term =.. *List*

?Term	<term>
?List	<list> or <variable>

List is a list whose first element is the principal functor of the *Term*, and whose tail is the list of arguments of *Term*.

If *Term* is an uninstantiated variable, then *List* must be instantiated to a list of determinate length. A compound term will be constructed from the list. The functor of the term will be the head of the list, while the arguments of the term will be the tail of the list.

If *Term* is not a variable, then the corresponding list is constructed and unified with *List*.

See Also
functor/3
arg/3

=/2

unification between two terms

Term1 = *Term2*

?Term1	<term>
?Term2	<term>

Succeeds if terms *Term1* and *Term2* unify. It is defined by the clause:

```
X = X.
```

See Also

\=/2
\==/2

=:=/2

expression equality

E1 =:= *E2*

+E1	<expr>
+E2	<expr>

Succeeds if the arithmetic expressions *E1* and *E2* both evaluate to the same value. Note that because of rounding errors, comparing two floating point numbers is not very reliable.

See Also

</2
=</2
>/2
>=/2
=\=/2
is/2

=</2

expression less than or equal

E1 =< E2

+E1	<expr>
+E2	<expr>

Succeeds if the value of the arithmetic expression *E1* is less than or equal to the value of the arithmetic expression *E2*.

See Also

</2
>/2
>=/2
=\=/2
=:=
is/2

==/2

check that two terms are identical

Term1 == Term2

?Term1	<term>
?Term2	<term>

Succeeds if term *Term1* is identical to *Term2*. No variables in *Term1* and *Term2* are bound as a result of the testing.

See Also

\==/2
@>/2
@>=/2
@</2
@=</2

=\=/2

expression inequality

E1 =\= E2

+E1	<expr>
+E2	<expr>

Succeeds if the arithmetic expressions *E1* and *E2* do not evaluate to the same value.

See Also

</2
=</2
>/2
>=/2
=:=
is/2

>/2

expression greater than

E1 > E2

+E1	<expr>
+E2	<expr>

Succeeds if the value of the arithmetic expression *E1* is greater than the value of the arithmetic expression *E2*.

See Also

</2
=</2
>=/2
=\=/2
=:=
is/2

>=/2

expression greater than or equal

E1 >= E2

+E1	<expr>
+E2	<expr>

Succeeds if the value of the arithmetic expression *E1* is greater than or equal to the value of the arithmetic expression *E2*.

See Also

</2
=</2
>/2
=\=/2
=:=
is/2

@</2

check that one term is less than another

Term1 @< *Term2*

?Term1	<term>
?Term2	<term>

Succeeds if *Term1* is less than *Term2* according to the standard ordering of terms. If the terms are of the same type a type comparison is made.

See Also

==/2
\==/2
@>/2
@>=/2
@=</2

@=</2

check that one term is equal to or less than another

Term1 @=< *Term2*

?Term1	<term>
?Term2	<term>

Succeeds if *Term1* is less than or identical to *Term2* according to the standard ordering of terms. If the terms are of the same type a type comparison is made.

See Also

==/2
\==/2
@>/2
@>=/2
@</2

@>/2

check that one term is greater than another

Term1 @> *Term2*

?Term1	<term>
?Term2	<term>

Succeeds if *Term1* is greater than *Term2* according to the standard ordering of terms. If the terms are of the same type a type comparison is made.

See Also

==/2
\==/2
@>=/2
@</2
@=</2

@>=/2

check that one term is greater than or equal to another

Term1 @>= *Term2*

?Term1	<term>
?Term2	<term>

Succeeds if *Term1* is greater than or identical to *Term2* according to the standard ordering of terms. If the terms are of the same type a type comparison is made.

See Also

==/2
\==/2
@>/2
@</2
@=</2

\+/1

negation

\+ *Call*

+Call <goal>

Negation as failure. This succeeds if *Call* fails. It is declared as a prefix operator.

Call can be a conjunction, or any other control term.

See Also

!/0
./2
->/2
:/2
abort/0
break/0
break_hook/1
fail/0
false/0
halt/0
halt/1
not/1
otherwise/0
repeat/0
repeat/1
true/0

\=/2

check for non-unification between two terms

Term1 \= *Term2*

?Term1	<term>
?Term2	<term>

The term *Term1* does not unify with the term *Term2*. It is defined as:

```
X \= Y :-
    \+ X = Y.
```

See Also

=/2
==/2
\==/2
@>=/2
@>/2
@</2
@=</2

\==/2

check that two terms are not identical

Term1 \== *Term2*

?Term1	<term>
?Term2	<term>

Succeeds if *Term1* is not identical to *Term2*.

See Also

==/2
\=/2
@>=/2
@>/2
@</2
@=</2

^/2

existential quantifier

X ^ *P*

?X	<term>
+P	<goal>

X ^ *P* can be read as "there exists an *X* such that *P* is true". The existential quantifier ^ is only of use when calling goals within *setof*/3 and *bagof*/3 where it has the effect of reducing the partitioning of sets due to free variables.

See Also
setof/3
bagof/3

~>/2

re-direct output to a file or a string

Call ~> Out

+Call	<goal>
?Out	<file_spec> or <variable>

If *Out* is a file specification, all output performed by the *Call* is re-directed to that file. If the *Call* is re-satisfiable, then on backtracking into the *Call* any further output is also redirected to that file. Note: that the file is not closed automatically.

If *Out* is a variable, all output performed by the *Call* is re-directed into a string, which is then unified with the variable *Out*. Note: in this mode, ~>/2 is not re-satisfiable even if the *Call* is.

~>/2 is defined as a non-associative infix operator.

See Also
<~/2

abolish/1

delete all the predicates specified by the given argument

abolish(*Rels*)

+Rels <pred_specs>

Abolishes all of the predicates specified by *Rels*.

Rels must be one of the following:

A predicate specification of the form *Name*/*Arity*, where *Name* is the name of a dynamic predicate and *Arity* is an integer. All clauses defining *Name* with *Arity* arguments will be deleted.

A list of predicate specifications, where each specification is in the form shown above. Each predicate on this list will be abolished in turn.

Please note: Modifying dynamic code while it is running can lead to unpredictable behaviour.

See Also

abolish/2
abolish_files/1
assert/1
assert/2
asserta/1
assertz/1
clause/2
clauses/2
clause/3
dynamic/1
functor/3
listing/0
listing/1
retract/1
retractall/1
retract/2
volatile/1

abolish/2

delete all clauses for the given predicate and arity

abolish(*Functor*, *Arity*)

+Functor	<functor>
+Arity	<arity>

Deletes all clauses for a given predicate. The *Functor* argument must be an atom that is the functor of the predicate to be deleted. The *Arity* argument must be a non-negative integer that is the number of arguments of the predicate to be deleted.

Please note: Modifying dynamic code while it is running can lead to unpredictable behaviour.

See Also

abolish/1
abolish_files/1
assert/1
asserta/1
assert/2
assertz/1
clause/2
clauses/2
clause/3
dynamic/1
functor/3
listing/0
listing/1
retract/1
retractall/1
retract/2
volatile/1

abolish_files/1

abolish all predicates associated with the given file

abolish_files(*FileSpec*)

+FileSpec <file_specs>

This *abolish_files*/1 predicate will abolish all the predicates associated with the given file *FileSpec*. *FileSpec* can be a file specification of the form: *PathAlias*(*File*) where *PathAlias* is an alias that refers to a specified path and *File* is a file name relative to that path. *FileSpec* can also be an atom of the form: *Path/File* where *Path* defines the path where the file *File* will be found.

See Also

compile/1
consult/1
ensure_loaded/1
initialization/1
load_files/1
load_files/2
multifile/1
prolog_load_context/2
reconsult/1
save_predicates/2
source_file/1
source_file/2
source_file/3

'?ABORT?'/0

user-defined Prolog program which is called following a program abort

User-defined program that is called after the Prolog system has aborted. The abort hook is called every time a call to the predicate *abort/0* is made. Note that *abort/0* is called by the default error hook. To allow the default post-processing of abort conditions you should call *abort_hook/0*.

See Also
abort_hook/0

abort/0

abort the current program

Abandons the program that is currently being executed and returns to the top level of Prolog. This predicate is normally only used when an error condition has occurred and there is no way of recovering.

After the program abort the predicate *abort*/0 passes control to the user-defined *'?ABORT?'/0* hook, if a definition is present, otherwise control goes to the *abort_hook/0* predicate.

See Also

!/0
,/2
->/2
;/2
\+/1
fail/0
false/0
halt/0
halt/1
not/1
otherwise/0
repeat/0
repeat/1
true/0
break/0
break_hook/1

abort_hook/0

built-in abort hook

Invoke the system defined abort hook. The predicate *abort_hook/0* prints the 'Aborted' message to the console and returns to the Prolog supervisor loop, this may be used in conjunction with user-defined *'?ABORT?'/0* programs.

See Also

!/0
./2
->/2
:/2
\+/1
abort/0
fail/0
false/0
halt/0
halt/1
not/1
otherwise/0
repeat/0
repeat/1
true/0
break/0
break_hook/1

absolute_file_name/2

converts from a relative to an absolute file specification

absolute_file_name(*RelFileSpec, AbsFileName*)

+RelFileSpec	<file_spec>
-AbsFileName	<variable>

RelFileSpec is a given relative file path and file name. *AbsFileName* will be bound to the given file's absolute file path and file name.

If an extension is given in *RelFileSpec*, *absolute_file_name/2* returns the absolute file name of *RelFileSpec* with that extension.

If no extension is given in *RelFileSpec*, *absolute_file_name*/2 finds *AbsFileName* by first making a search for a file with a '.PL' extension, if that file exists then *absolute_file_name*/2 returns the absolute file name of *RelFileSpec* with a '.PL' extension, otherwise the absolute file name of *RelFileSpec* without an extension is returned.

See Also

absolute_file_name/3
cat/3
close/1
fclose/1
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
attrib/2
chdir/1
del/1
dir/3
drive/1
env/2
fcreate/3
ren/2
rmdir/1
stamp/1

absolute_file_name/3

convert between a relative and an absolute file specification using options

absolute_file_name(*RelFileSpec*, *Options*, *AbsFileName*)

+RelFileSpec	<file_spec>.
+Options	<list_of <compound_term> >
-AbsFileName	<variable>.

Find an absolute path name using the given relative file specification and options. The *RelFileSpec* argument is a relative file path including a file name. The *Options* argument is a list of zero or more of the following tuples: ignore_underscores(*Bool*), extensions(*Ext*), file_type(*Type*), access(*Mode*). The *AbsFileName* argument will be bound to the absolute file path and file name of the given files.

The predicate *absolute_file_name*/3 works as a four phase process, in which each phase gets a possible solution from the preceding phase, and constructs one or more alternate solutions to be tested by the succeeding phases. The phases are:

1. Syntactic rewriting
2. Underscore deletion
3. Extension expansion
4. Access checking

Each of the first three phases modifies the possible solution passed to it and produces alternatives that will be fed into the succeeding phases. The functionality of all phases but the 'Syntactic rewriting' phase are decided with the option list. The 'Access checking' phase checks if the generated file exists, and if not asks for a new alternative from the preceding phases. If the file exists, but doesn't obey the access mode option, a permission exception is raised. If the file obeys the access mode option, then *absolute_file_name*/3 commits to that solution, unifies *AbsFileName* with the file name, and succeeds determinately.

Note that the relative file name argument *RelFileSpec* may also be of the form *PathAlias(FileSpec)*, in which case the absolute file name of the file *FileSpec* in one of the directories designated by *PathAlias* is returned.

See Also

absolute_file_name/2
cat/3
close/1
fclose/1
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
attrib/2
chdir/1
del/1
dir/3
drive/1
env/2
fcreate/3

ren/2
rmdir/1
stamp/1

absolute_file_name/3 - Syntactic Rewriting Phase

this phase translates the relative file specification given by *RelFileSpec* into the corresponding absolute file name.

If *RelFileSpec* is a term with one argument, it is interpreted as: <path_alias>(<file_spec>) and outfile becomes the file as given by *file_search_path*/2. If *file_search_path*/2 has more than one solution, outfile is unified with the solutions in the order they are generated. If the succeeding phase fails, and there are no more solutions, an existence exception is raised.

If *RelFileSpec* = '', outfile is unified with the current working directory. If *absolute_file_name*/3 is called from a goal in a file being loaded, the directory containing that file is considered current working directory. If the succeeding phase fails, an existence exception is raised.

If *RelFileSpec* is an atom, other than '', it's divided into components. A component is defined to be those characters:

1. Between the beginning of the file name and the end of the file name if there are no '\'s in the file name.

2. Between the beginning of the file name and the first '\'.

3. Between any two successive '\'-groups (where a '\'-group is defined to be a sequence of one or more '\'s with no non-'\' character interspersed.)

4. Between the last '\' and the end of the file name.

To give the absolute file name, the following rules are applied to each component of RelFileSpec:

1. *RelFileSpec* is prefixed with the current working directory. If *absolute_file_name*/3 is called from a goal in a file being loaded, the directory containing that file is considered the current working directory.

2. The component '.' is deleted.

3. The component '..' is deleted together with the directory name syntactically preceding it. For example, 'a/b/../c' is rewritten as 'a/c'.

4. Two or more consecutive '/'s are replaced with one '/'.

When these rules have been applied, the absolute file name is unified with outfile. If the succeeding phase fails, an existence exception is raised.

absolute_file_name/3 - Underscore deletion phase

If the option list includes the ignore_underscores(true) option this phase constructs two names: first the file name that is derived *directly* from the output of the previous phase, and then the file name obtained by first deleting all the underscores from the output of the previous phase.

absolute_file_name/3 - Extension expansion phase

This phase is dependent on whether the *RelFileSpec* contains an extension or not and the extensions and file_type options.

absolute_file_name/3 - Access checking phase

This is dependent on the setting of the access option.

absolute_file_name/3 Option - ignore_underscores(Bool)

Where *Bool* is one of the following:

true When constructing an absolute file name that matches the given access modes, two names are tried: First the absolute file name derived directly from *RelFileSpec*, and then the file name obtained by first deleting all underscores from *RelFileSpec*.

false (default) Suppresses any deletion of underscores.

absolute_file_name/3 Option - extensions(Ext)

Has no effect if *RelFileSpec* contains a file extension. *Ext* is an atom or a list of atoms, each atom representing an extension that should be tried when constructing the absolute file name. The extensions are tried in the order they appear in the list. Default value is *Ext* = [''], i.e. only the given *RelFileSpec* is tried, no extension is added. To specify extensions('') or extensions([]) is equivalent to not giving any extensions option at all.

absolute_file_name/3 Option - file_type(Type)

Has no effect if *RelFileSpec* contains a file extension. Picks an appropriate extension for the operating system currently running, which means that programs using this option instead of extensions(*Ext*) will be more portable on most systems. *Type* must be one of the following atoms:

text implies *extensions([''])*. *RelFileSpec* is a file without any extension. (Default)

Prolog implies *extensions(['','pl'])*. RelFileSpec is a Prolog source file, maybe with a .pl extension.

qof implies *extensions(['','pc'])*. RelFileSpec is a Prolog object code file, maybe with a .qof extension.

absolute_file_name/3 Option - access(Mode)

Mode must be one of the following atoms:

read *AbsFileName* must have read access.

write If *AbsFileName* exists, it must have write access. If it doesn't exist, it must be possible to create.

append If *AbsFileName* exists, it must have write access. If it doesn't exist, it must be possible to create.

exist The file represented by *AbsFileName* must exist.

none (default) If an extensions option is specified, this is used in a search for an actual file. If a file with extension exists, the absolute file name of that file (including extension) is returned, otherwise the absolute file name of the *RelFileSpec* without any extension (unless *RelFileSpec* itself has an extension specified), is returned. If no extensions option is present, the file system is not accessed. The first absolute file name that is derived from *RelFileSpec* is returned. Note that if this option is specified, no existence exceptions can be raised.

abtbox/3

display the about box

abtbox(*Title*, *Message*, *Font*)

+Title	<string>
+Message	<atom> or <string>
+Font	<atom> or <integer> in the domain {0,1}

Displays the "about" dialog box with the given title (window caption) and message, using the given font.

The predicate succeeds if the 'OK' button is clicked or the <return> key is pressed, or fails if the dialog is closed with 'Close' system menu option or the <escape> key is pressed.

See Also

chgbox/3
change_hook/3
dirbox/4
erase_status_box/0
find_hook/3
fndbox/2
message_box/3
msgbox/4
status_box/1

sttbox/2

ansoem/2

convert between ansi and oem strings

ansoem(*Ansi*,*Oem*)

?Ansi	<string>, <atom> or <variable>
?Oem	<string>, <atom> or <variable>

Convert the *Ansi* string to *Oem* format, or vice versa. The input may be a string or atom; the type of the output depends upon the type of the input. Note that because the ANSI and OEM character sets are not congruent, the conversion is not necessarily reversible. For this reason, the case where both strings are given is not supported.

Normally the only font in Windows which uses the OEM character set is the IBM PC font (font number 0 - see *wfont/2*) other Windows fonts generally use the ANSI character set.

See Also
fonts/1
wfclose/1
wfcreate/4
wfdata/5
wfdict/1
wfont/2
wfsize/4

append/3

join or split lists

append(*First*, *Second*, *Whole*)

?First	<list> or <variable>
?Second	<list> or <variable>
?Whole	<list> or <variable>

If *First* and *Second* are instantiated, the *Second* list is appended to the end of the *First* list to give the *Whole* list

```
?- append([a,b,c,d], [1,2,3,4], Whole ).
Whole = [a,b,c,d,1,2,3,4].
```

If *Whole* is instantiated, the *Whole* list is split at an arbitrary place to form two new sub-lists *First* and *Second*. Alternative solutions will be provided on backtracking.

```
?- append(First, Second, [a,b,c,d,1,2,3,4] ).
First = []
Second = [a,b,c,d,1,2,3,4];

First = [a]
Second = [b,c,d,1,2,3,4].
```

If *First* and *Whole* are instantiated and the *Whole* list starts with the *First* list, then the remainder of the *Whole* list is returned as *Second*.

```
?- append([a,b,c,d], Second, [a,b,c,d,1,2,3,4] ).
Second = [1,2,3,4].
```

If *Second* and *Whole* are instantiated and the *Whole* list ends with the *Second* list then the beginning of the *Whole* list is returned as *First*.

```
?- append(First, [1,2,3,4], [a,b,c,d,1,2,3,4] ).
First = [a,b,c,d].
```

If *First*, *Second* and *Whole* are instantiated, then test that the *Whole* list begins with the *First* list and that the remainder of the *Whole* list is the *Second* list.

```
?- append([a,b,c,d], [1,2,3,4], [a,b,c,d,1,2,3,4] ).
yes.
```

See Also

length/2
mem/3
member/2
member/3
remove/3
removeall/3
reverse/2

arg/3

find the nth argument of a term

arg(*N*, *Term*, *Arg*)

+N	<integer>	0
+Term	<term>	
-Arg	<variable>	

Unifies *Arg* with the *Nth* argument of *Term*. *N* must be a positive integer, and *Term* must be a compound term (or a list). The arguments are numbered 1 upwards. Lists are considered compound terms of arity 2.

See Also

=../2
call/1
call/2
functor/3
mem/3
one/1

assert/1

add a clause at the end of the clauses associated with its predicate name

assert(*Clause*)

+Clause <clause>

Adds *Clause* to the database at the end of the sequence of clauses defining its predicate name. *Clause* must be a "fact" or a "rule". Facts should be given in the form:

```
atom
atom(arg1, _, argk)                    k > 1
```

where *atom* is a Prolog atom.

Rules should be given in the form:

```
(Head :- C1, C2, _, Cn)                n > 1
```

where *Head*, *C1*,*C2*,, _, and *Cn* are atoms or compound structures.

Note: the rule *must* be enclosed in parentheses because the operator precedence of *:-*/2 is greater than 1000.

Any unbound variable found in *Clause* will be treated as a variable in the asserted clause. You cannot assert a clause defining a predicate if that predicate is already defined to be static.

See Also

abolish/1
abolish/2
abolish_files/1
assert/2
asserta/1
assertz/1
clause/2
clauses/2
clause/3
dynamic/1
functor/3
listing/0
listing/1
retract/1
retractall/1
retract/2
volatile/1

assert/2

assert the clause at the given position

assert(*Clause*, *Position*)

+Clause	<clause>	
+Position	<integer>	0

Asserts the *Clause* at the specific *Position* within the clauses for that predicate (given as the principal functor in the *Clause* head). If the *Position* is 0 then it is added at the end (*assertz*/1). If the *Position* is 1 then it is added at the beginning (*asserta*/1). If the *Position* is greater than the current number of clauses for that predicate then it is added at the end (*assertz*/1). Otherwise, the clause is inserted at the given *Position* relative to the beginning.

See Also

abolish/1
abolish/2
abolish_files/1
assert/1
asserta/1
assertz/1
clause/2
clauses/2
clause/3
dynamic/1
functor/3
listing/0
listing/1
retract/1
retractall/1
retract/2
volatile/1

asserta/1

add a clause at the beginning of the clauses associated with its predicate name

asserta(*Clause*)

+Clause <clause>

As *assert*/1 except that *Clause* is added at the beginning of the sequence of clauses that define its predicate name.

See Also

abolish/1
abolish/2
abolish_files/1
assert/2
assert/1
assertz/1
clause/2
clauses/2
clause/3
dynamic/1
functor/3
listing/0
listing/1
retract/1
retractall/1
retract/2
volatile/1

assertz/1

add a clause at the end of the clauses associated with its predicate name

assertz(*Clause*)

+Clause <clause>

Adds *Clause* to the database at the end of the sequence of clauses defining its predicate name. This predicate is exactly the same as *assert/1*.

See Also

abolish/1
abolish/2
abolish_files/1
assert/2
assert/1
asserta/1
clause/2
clauses/2
clause/3
dynamic/1
functor/3
listing/0
listing/1
retract/1
retractall/1
retract/2
volatile/1

at_end_of_file/0

test to see if the input file pointer is at end of file

The predicate *at_end_of_file*/0 checks if end of file has been reached for the current input stream. An input stream reaches end of file when all characters except the end of file marker have been read. It remains at end of file after the end of file marker has been read.

See Also

at_end_of_line/0
find/1
inpos/1
outpos/1
skip/1
skip_layout/0
skip_line/0
skip_term/0
stream_position/2
stream_position/3
flush/0

at_end_of_line/0

test whether end of line has been reached for the current input stream.

at_end_of_line/0 succeeds when the end of line marker is reached for the current input stream. An input stream reaches end of line when all the characters except the line border of the current line have been read. *at_end_of_line*/0 is also true whenever *at_end_of_file*/0 is true.

Coding with *at_end_of_line*/0 to check for end of line is more portable among different operating systems than checking end of line by the input character code.

See Also

at_end_of_file/0
find/1
inpos/1
outpos/1
skip/1
skip_layout/0
skip_line/0
skip_term/0
stream_position/2
stream_position/3
flush/0

atom/1

test for an atom

atom(*Term)*

?Term	<term>

Succeeds if, and only if, *Term* is instantiated to a Prolog atom (including the special atom *[]*).

See Also

atomic/1
callable/1
char/1
chars/1
compound/1
float/1
ground/1
integer/1
integer_bound/3
nonvar/1
number/1
simple/1
string/1
type/2
unifiable/2
var/1

atom_chars/2

converts between an atom and a list of characters

atom_chars(*Atom, CharList*)

?Atom	<variable> or <atom>
?CharList	<char_list> or <variable>

Initially either *CharList* must be instantiated to a list of ASCII character codes (containing no variables) or *Atom* must be instantiated to an atom.

If *Atom* is initially instantiated to an atom, *CharList* will be unified with the list of ASCII character codes that represent its printed representation. If *CharList* is initially instantiated to a list of ASCII character codes then *Atom* will be unified with an atom containing those characters.

See Also

atom_string/2
string_chars/2
name/2
number_atom/2
number_chars/2
number_string/2

atom_string/2

convert between an atom and a string

atom_string(*Atom*, *String*)

?Atom	<atom> or <variable>
?String	<string> or <variable>

Convert between "atom" and "string" data-types. Initially either *Atom* must be instantiated to an atom or *String* must be instantiated to a string. If *Atom* is initially instantiated, *String* will be unified with the string type equivalent of that atom. If *String* is initially instantiated then *Atom* will be unified with an atomic equivalent of that string.

See Also

atom_chars/2
string_chars/2
name/2
number_atom/2
number_chars/2
number_string/2

atomic/1

test for an atom, string or number

atomic(*Term)*

?Term	<term>

Test for atomic data, in the sense that the data cannot be broken into sub-structures. Succeeds if the *Term* argument is currently instantiated to an atom, string, or a number. Fails for any other type of term (a compound term or a list).

See Also

atom/1
callable/1
char/1
chars/1
compound/1
float/1
ground/1
integer/1
integer_bound/3
nonvar/1
number/1
simple/1
string/1
type/2
unifiable/2
var/1

attrib/2

set or get file attributes

attrib(*Filename*, *Attrib*)

+Filename	<atom>
?Attrib	<integer> or <variable>

Set or get the read and hide attributes for the file specified by the *Filename* argument. If the *Attrib* argument contains an attribute value then the file attributes will be set accordingly. If *Attrib* is a variable it will be bound to the currently set file attribute values.

These attributes may be set or checked for ordinary files and directories, but not for system files or the volume label. Setting any of these attributes clears the archive bit.

See Also

absolute_file_name/2
absolute_file_name/3
cat/3
close/1
fclose/1
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
attrib/2
chdir/1
del/1
dir/3
drive/1
env/2
fcreate/3
ren/2
rmdir/1
stamp/1

bagof/3

find all the instances of a term for which a Prolog goal is true

bagof(*Term*, *Call*, *List*)

?Term	<term>
+Call	<goal>
?List	<variable> or <list>

Succeeds if *List* is a non-empty list of instances of *Term* such that *Call* is provable. *Term* may be any type of Prolog term, *Call* must be a call of the form:

```
V1^V2^...^Vn^Goal                    (n _ 0)
```

where *V1*,*V2*,…,*Vn* are variables in the call to solve *Goal*. They are the existentially quantified variables of *Goal*.

The solution list with which *List* is unified will not be sorted and may contain duplicate entries. The predicate *bagof*/3 is similar to *setof*/3 and differs only in that the solution list is not sorted and may contain duplicate entries.

See Also

^/2
findall/3
forall/2
setof/3
solution/2

beep/2

sound a beep of the given duration and frequency

beep(*Frequency*, *Duration*)

+Frequency	<integer> in the range [0,20..20000]
+Duration	<integer>

Sounds a beep on the internal speaker of the computer with the given *Frequency* and *Duration*.

Frequency must be 0 or an integer in the range 20 to 20000. It specifies the frequency of the beep in Hertz. A frequency of 0 gives a silent pause for the specified duration. A frequency of 261Hz corresponds to middle C on a piano.

Duration must be 0 or an integer. If *Duration* is an integer it specifies the number of milliseconds the beep will last. A duration of 0 sounds the beeper for an indefinite period until another call to beep is made with a frequency of 0. Any beep can be interrupted by hitting any key.

'?BREAK?'/1

user-defined Prolog program which intercepts break messages

'?BREAK?'(*Goal*)

+Goal <goal>

User-defined program to intercept Prolog break messages. The *Goal* argument is the goal that was broken into. The one error which is not handled by the error hook, is error 11, this is generated whenever the <ctrl-break> key is hit. This key is used to break into a Prolog program. To allow the default processing of break messages you should call *break_hook/1*.

See Also
break_hook/1

break/0

suspend the current execution

break/0 causes the current execution to be interrupted and displays the message

```
[ Break level 1 ]
?-
```

The system is then ready to accept input as though it were at the top level. If another call to *break*/0 is encountered it moves up to level 2, and so on. The break level is displayed on a separate line before each top-level prompt.

```
[1]
?-
```

To close a break level and resume the execution which was suspended, type the end of file character (<esc>). *break*/0 then succeeds, and execution of the interrupted program is resumed. Alternatively, the suspended execution can be terminated by calling *abort*/0.

See Also

!/0
./2
->/2
:/2
\+/1
abort/0
fail/0
false/0
halt/0
halt/1
not/1
otherwise/0
repeat/0
repeat/1
true/0
break_hook/1

break_hook/1

built-in break hook

break_hook(*Goal*)

?Goal <term>

Invoke the system defined program break hook with the given *Goal*. The predicate *break_hook/1* displays a dialog showing that a program break has been requested and then asks if you want to continue with the given *Goal*. If you choose to continue the *Goal* will be run. If you choose not to continue the *Goal* will not be run and you will be returned to the Prolog supervisor. This predicate is mainly provided to allow programmatic access to the default system break handler in user-defined *'?BREAK?'/1* programs.

See Also

!/0
,/2
->/2
;/2
\+/1
abort/0
fail/0
false/0
halt/0
halt/1
not/1
otherwise/0
repeat/0
repeat/1
true/0
break/0

'C'/3

used in the expansion of grammar rules

'C'(*S1*, *Terminal*, *S2*)

?S1	<list> or <variable>
?Terminal	<term>
?S2	<list> or <variable>

The *'C'/3* predicate is not of direct use to the user. It is used by Prolog when expanding grammar rules that contain terminal symbols. For more details see the section on the Prolog representation of grammar rules.

The definition of this predicate is:

```
'C'([Terminal|Rest], Terminal, Rest).
```

See Also

expand_term/2
phrase/2
phrase/3

call/1

call a Prolog goal.

call(*Call*)

+Call <goal>

Calls the goal *Call*. Succeeds if *Call* succeeds, and fails otherwise. If *Call* is bound to *!*, or a call that contains a *!*, the cut will only cut out alternatives in *Call*. For example in the following program:

```
test :-
        call(!).
test.
```

the call to *!* will *not* cut out backtracking to the second clause for test: it will be local to the call itself.

Compare this behaviour with the following use of a call meta-variable:

```
test :-
        X = !,
        X.
test.
```

In this example, the meta call in the first clause will cut out the second clause for test.

See Also

=../2
arg/3
call/2
functor/3
one/1

call/2

call a Prolog goal and return the termination port

call(*Call*, *Port*)

+Call	<goal>
-Port	<variable>

call/2 always succeeds and returns the port at which the call terminates. *Port* will be bound to one of the following:

exit	A solution, with a possibility of more.
done	A solution, with no more solutions.
fail	No solutions.

See Also

=../2
arg/3
call/1
functor/3
one/1

call_dialog/2

call a modal dialog and get or check the result

call_dialog(*Window*, *Result*)

+Window	<window_handle>
?Result	<term>

Shows the given dialog *Window*, disables all windows created by Prolog (apart from the dialog *Window* and its children) and runs the dialog. The predicate *call_dialog/2* succeeds when the window's handler returns a result that matches with the given *Result* or fails when the window's handler returns a result that does not match with the given *Result*. *Window* should be the name of a previously created dialog window.

See Also

window_handler/2
window_handler/4
show_dialog/1
wdcreate/7

callable/1

test to see if a term is an atom or a compound.

callable(*Term*)

+Term <term>

callable/1 succeeds if *Term* is currently instantiated to a term that *call*/1 would take as an argument and not give a type error (an atom or a compound term); otherwise it fails.

See Also
atom/1
atomic/1
char/1
chars/1
compound/1
float/1
ground/1
integer/1
integer_bound/3
nonvar/1
number/1
simple/1
string/1
type/2
unifiable/2
var/1

cat/3

atom and string concatenation

cat(*Parts*, *Whole*, *Joins*)

?Parts	<list> or <variable>
?Whole	<string> or <atom> or <variable>
?Joins	<list> or <variable>

This predicate can be used to join an arbitrary number of strings or atoms together, or to split one into an arbitrary number of parts.

To join strings or atoms: *Parts* must be bound to a list of atoms or a list of strings; *Whole* should be a variable that will return the resulting string or atom, and *Joins* should be a variable that will contain a list describing where the joins were made.

To split a string or atom: *Whole* should be bound to the string or atom to be split; *Joins* should be bound to a list describing where the splits should be made, and *Parts* should be a variable that will become the list of strings or atoms obtained by splitting the input at the specified points.

See Also

<~/2
~>/2
cat/3
elex/1

catch/2

catch the error code generated by the given goal

catch(*Error*, *Goal*)

-Error	<variable>
+Goal	<goal>

Catch the *Error* number generated by running the given *Goal.* If the *Goal* succeeds an *Error* number of 0 is returned. If the *Goal* fails an *Error* number of -1 is returned. If the *Goal* generates an error, *Error* is bound to the number of the error. This predicate can work in conjunction with *throw/2,* which is used to declare that a given goal has generated a given numbered error.

If an error is generated in a goal and there is no call to *catch/2* remaining on the call stack, the goal that generated the error and the error number is passed to the user-defined *'?ERROR?'/2,* if it exists, or otherwise to the default sytem *error_hook/2*.

Warning: *catch/2* always succeeds on first call, whether the given *Goal* succeeds, fails or generates an error. Because of this, it has to introduce a choice point on the backtrack stack, even if *Goal* is deterministic. Thus:

```
?- catch(Code,true).
```

will return Code = 0, but leave a choice point on the stack. If you request a reevaluation, it will return Code = -1. Any subsequent reevaluation would cause *catch/2* to fail. When inserting *catch/2* into programs, you should bear these additional choice points in mind. You can add a call to cut (*!/0*) to the *Goal* you catch if you want the call to be deterministic, eg:

```
?- catch(Code,(true,!)).
```

will return Code = 0, but will not leave any choice points, and will not allow *catch/2* to backtrack and generate the -1 solution.

If the given *Goal* fails or generates an error, this removes any choice point introduced by *catch/2* itself. Thus, once a non-zero error code has been returned, it is safe to assume that there are no longer any choice points remaining on the stack due to that particular call to *catch/2*.

See Also

abort/0
error_message/2
unknown_predicate_handler/2
catch/3
throw/2
error_hook/2
flush/0

catch/3

same as catch/2 but return the predicate that actually generated the error

catch(*Error*, *Goal*, *ErrorPred*)

-Error	<variable>
+Goal	<goal>
-ErrorPred	<variable>

Catch the *Error* number generated during the running of the given *Goal* and return the functor and arity of the goal that actually generated the error. If the *Goal* succeeds an *Error* number of 0 is returned and *ErrorPred* remains unbound. If the *Goal* fails an *Error* number of -1 is returned and *ErrorPred* is bound to the functor and arity of the last goal that failed during the evaluation of the *Goal*. If an error is generated during the running of the *Goal*, *Error* is bound to the number of the error and *ErrorPred* is bound to the functor and arity of the predicate that actually generated the error.

Warning: like *catch/2*, *catch/3* always succeeds on first call, whether the given *Goal* succeeds, fails or generates an error.

See Also

abort/0
error_message/2
unknown_predicate_handler/2
catch/2
throw/2
error_hook/2
flush/0

'?CHANGE?'/3

user-defined hook for handling change box messages

'?CHANGE?'(*Focus*, *Message*, *Goal*)

+Focus	<window_handle>
+Message	<integer>
+Goal	<goal>

Where *Focus* is the window that is currently in focus, *Message* is the message that was generated and *Goal* is the goal that was interrupted by the *Message*.

In response to one of the change box messages, you should make a call to *chgbox/3* this time giving three variables. The first two of these will return the text in the "from" and "to:" boxes as strings, and the third will return an integer in the range 1 to 3 indicating the radio button choice:

Once this data has been obtained, the dialog should be reenabled by yet another call to *chgbox/3*, this time with the strings/integer argument trio.

```
?- chgbox(`hello`,``,1).
```

This will display and enable the modeless find dialog box with the word 'hello' in the first "edit" field and whatever is in its current second "edit" field, returning immediately.

You can invoke the default message handler directly by calling the predicate *change_hook/3*.

See Also
change_hook/3
chgbox/3

change_hook/3

system handler for the change dialog

change_hook(*Win, Msg, Goal*)

+Win	<window_handle>
+Msg	<integer>
+Goal	<term>

Invoke the system behaviour for the change dialog in the given window (*Win*) for the given message (*Msg*) then run the suspended *Goal*. *change_hook/3* reacts to the messages sent by the change dialog and carries out the requested behaviour.

See Also

abtbox/3
chgbox/3
dirbox/4
erase_status_box/0
find_hook/3
fndbox/2
message_box/3
msgbox/4
status_box/1
sttbox/2

char/1

test for an integer that represents a character

char(*Term*)

+Term	<char>

Succeeds if *Term* is an ASCII character code (i.e. *Term* is an integer in the range 0 =< *Term* =< 255).

See Also

atom/1
atomic/1
callable/1
chars/1
compound/1
float/1
ground/1
integer/1
integer_bound/3
nonvar/1
number/1
simple/1
string/1
type/2
unifiable/2
var/1

chars/1

test for a list of integers that all correspond to characters

chars(*Term*)

+Term	<char_list> in the range [0..255]

Succeeds if *Term* is a complete list of ASCII character codes (i.e. each member of the list is instantiated to an integer, I, that is in the range 0 =< I =< 255).

See Also

atom/1
atomic/1
callable/1
char/1
compound/1
float/1
ground/1
integer/1
integer_bound/3
nonvar/1
number/1
simple/1
string/1
type/2
unifiable/2
var/1

chdir/1

choose or return a directory

chdir(*Dir*)

?Dir	<atom> or <variable>

Finds or changes the current directory. The current directory is where Prolog (and DOS) searches for a file when no directory is specified as part of the file name. There is a current directory associated with each disk drive.

If *Dir* is an unbound variable, *chdir*/1 binds it to the path name of the current directory on the current drive. If *Dir* is an atom, *chdir*/1 attempts to make *Dir* the current directory. If no drive is specified, the current drive is assumed.

See Also

absolute_file_name/2
absolute_file_name/3
cat/3
close/1
fclose/1
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
attrib/2
chdir/1
del/1
dir/3
drive/1
env/2
fcreate/3
ren/2
rmdir/1
stamp/1

chgbox/3

display the modeless change box

chgbox(*From*, *To*, *Flag*)

?From	<string> or <variable>
?To	<string> or <variable>
?Flag	<integer> or <variable>

Displays or hides the modeless "change" dialog box with the given prefill from and to strings, or returns data from the existing box. This dialog includes two "edit" fields, labeled "From:" and "To:", and three radio buttons, with which the user can select the scope of the find or change.

The *From* argument is a string that denotes the contents of the From: edit field. If an empty string is specified, the existing contents of the From: field are not altered. The *To* argument is a string that denotes the contents of the To: edit field. Like the *From* argument, if an empty string is specified, the existing contents of the To: field are not altered. The *Flag* argument is an action value that specifies an action to be performed on the change box.

The predicate succeeds if the change box is created properly, and returns control immediately. When the user later clicks one of the option buttons, the box is disabled and a message sent to Prolog giving details of the user's action.

Normally when the user selects an action in the change box the default message handler will be called. If you want to control the the behaviour of the change box yourself you will need to define a *'?CHANGE?'/3* program.

See Also

abtbox/3
change_hook/3
chgbox/3
dirbox/4
erase_status_box/0
find_hook/3
fndbox/2
message_box/3
msgbox/4
status_box/1
sttbox/2
'?CHANGE?'/3

clause/2

get or check the body of a clause given its head

clause(*Head*, *Body*)

+Head	<atom> or <compound_term>
?Body	<conjunct_of <clause> > or <variable>

Succeeds if there is a clause in the database whose head matches *Head* and whose body matches *Body*. The predicate of *Head* must be dynamic.

clause/2 searches through the database until it finds the first clause whose head matches *Head*. The body of this clause will be unified with *Body*. If the matching clause is a fact, *Body* will be unified with the single goal *true*.

The *clause*/2 predicate is non-deterministic. That is, it can be used to backtrack through all the clauses in the database that match a given *Head* and *Body*. It will fail when there are no (more) matching clauses.

See Also

abolish/1
abolish/2
abolish_files/1
assert/2
assert/1
asserta/1
assertz/1
clauses/2
clause/3
dynamic/1
functor/3
listing/0
listing/1
retract/1
retractall/1
retract/2
volatile/1

clause/3

find the position of a clause in a dynamic predicate

clause(*Head*, *Body*, *Posn*)

+Head	<compound_term> or <atom> representing clause head	
-Body	<variable>	
?Posn	<variable> or <integer>	0

Search the clauses of a dynamic predicate for one which unifies with *Head* returning the corresponding clause *Body*. The search starts with the first clause and if successful returns the position of the matching clause as *Posn*.

See Also

abolish/1
abolish/2
abolish_files/1
assert/1
asserta/1
assert/2
assertz/1
clause/2
clauses/2
dynamic/1
functor/3
listing/0
listing/1
retract/1
retractall/1
retract/2
volatile/1

clauses/2

return a list of candidates for a dynamic predicate that match a head

clauses(*Head*, *Clauses*)

+Head	<compound_term> representing clause head
-Clauses	<variable>

Returns the list of candidate *Clauses* for a dynamic predicate specified by *Head*. Candidate clauses are those whose 1st argument could possibly unify with the 1st argument of *Head*.

See Also

abolish/1
abolish/2
abolish_files/1
assert/1
asserta/1
assert/2
assertz/1
clause/2
clause/3
dynamic/1
functor/3
listing/0
listing/1
retract/1
retractall/1
retract/2
volatile/1

close/1

close the named file

close(*FileSpec*)

+FileSpec <file_spec>

If *FileSpec* refers to the name of an open file, the file buffer is written to disk and the file closed.

FileSpec can be a file specification of the form: *PathAlias*(*File*) where *PathAlias* is an alias that refers to a specified path and *File* is a file name relative to that path. *FileSpec* can also be an atom of the form: *Path/File* where *Path* defines the path where the file *File* will be found.

The *told*/0 and *seen*/0 predicates will do an automatic close if the current output stream is a file.

See Also

absolute_file_name/2
absolute_file_name/3
cat/3
fclose/1
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
attrib/2
chdir/1
del/1
dir/3
drive/1
env/2
fcreate/3
ren/2
rmdir/1
stamp/1

cmp/3

compare two terms

cmp(*Order*, *First*, *Second*)

?Order	<integer> in the domain {-1,0,1} or <variable>
+First	<term>
+Second	<term>

Get or check the *Order* of two terms *First* and *Second*. The values of *Order* that are returned or checked are: -1 if *First* < *Second*, 0 if *First* = *Second* and 1 if *First* > *Second*, according to the standard ordering of terms. When comparing the equality of the terms (i.e. *Order* is instantiated to 0) a check is made to see if the variables in the terms are identical.

The order is found according to the standard ordering of terms. If the terms are of the same type a type comparison is made.

See Also

=/2
==/2
@</2
@=</2
@>/2
@>=/2
\=/2
\==/2
compare/3
eqv/2
keysort/2
len/2
sort/2
sort/3
occurs_chk/2
subsumes_chk/2

compare/3

find the relationship between one term and another

compare(*Rel*, *Term1*, *Term2*)

?Rel	<atom> in the domain {=,<,>} or <variable>
?Term1	<term>
?Term2	<term>

Succeeds if *Rel* is the relationship between *Term1* and *Term2*.

The order is found according to the standard ordering of terms. If the terms are of the same type a type comparison is made.

If *Rel* is an unbound variable, it will be bound to =, <, or > according to the relationship between *Term1* and *Term2*.

If *Rel* is given, it must be one of =, <, >. *compare*/3 will succeed if *Rel* describes the relationship between *Term1* and *Term2*.

See Also

=/2
==/2
@</2
@=</2
@>/2
@>=/2
\=/2
\==/2
cmp/3
eqv/2
keysort/2
len/2
sort/2
sort/3
occurs_chk/2
subsumes_chk/2

compile/1

compile the specified Prolog source file(s) into object code format

compile(*FileSpec*)

+FileSpec <file_specs>

compile/1 loads the source file specified by *FileSpec* and compiles it into object code format using the optimising compiler.

FileSpec can be a file specification of the form: *PathAlias*(*File*) where *PathAlias* is an alias that refers to a specified path and *File* is a file name relative to that path. *FileSpec* can also be an atom of the form: *Path/File* where *Path* defines the path where the file *File* will be found.

FileSpec may also be a list of file specifications.

If *FileSpec* does not specify a file extension then '.PL' is assumed.

Any commands in the source file referred to by *FileSpec* are executed as they are encountered. A command is a term with functor *':-'*.

See Also

abolish_files/1
consult/1
ensure_loaded/1
initialization/1
load_files/1
load_files/2
multifile/1
prolog_load_context/2
reconsult/1
save_predicates/2
source_file/1
source_file/2
source_file/3

compound/1

test for a compound term

compound(*Term*)

?Term	<term>

Succeeds if *Term* is a compound term and fails otherwise.

See Also

atom/1
atomic/1
callable/1
char/1
chars/1
float/1
ground/1
integer/1
integer_bound/3
nonvar/1
number/1
simple/1
string/1
type/2
unifiable/2
var/1

consult/1

load a source code program into memory

consult(*FileSpec*)

+FileSpec <file_specs>

consult/1 loads the source file specified by *FileSpec*. *FileSpec* can be a file specification of the form: *PathAlias*(*File*) where *PathAlias* is an alias that refers to a specified path and *File* is a file name relative to that path. *FileSpec* can also be an atom of the form: *Path/File* where *Path* defines the path where the file *File* will be found.

If *FileSpec* does not specify a file extension then '.PL' is assumed.

Any commands in *FileSpec* are executed as they are encountered. A command is a term with functor *':-'*.

If *FileSpec* is the atom *user*, clauses and commands are entered from the terminal using the built-in editor.

When using the Prolog interactive environment described in the user guide, *consult(File)* can also be written as:

```
?- [File].
```

See Also

abolish_files/1
compile/1
ensure_loaded/1
initialization/1
load_files/1
load_files/2
multifile/1
prolog_load_context/2
reconsult/1
save_predicates/2
source_file/1
source_file/2
source_file/3

copy/2

copy data from the current input stream to the current output stream

copy(*Wanted, Copied*)

+Wanted	<integer>
-Copied	<variable>

Tries to copy the specified number of bytes, *Wanted*, from the current input stream to the current output stream. When the process has finished the number of bytes actually copied is returned as *Copied*.

See Also

close/1
fclose/1
fdict/1
fname/4
fopen/3
open/2
fcreate/3

copy_term/2

copy a term with new variables

copy_term(*Term*, *Copy*)

+Term	<term>
-Copy	<variable>

Term is a given term and *Copy* is unified with a copy of *Term* in which all variables have been replaced with new distinct variables.

See Also

=../2
atom_chars/2
atom_string/2
string_chars/2
name/2
number_atom/2
number_chars/2
number_string/2
numbervars/3

current_atom/1

check or get a current atom

current_atom(*ToTest*)

?ToTest	<atom> or <variable>

current_atom/1 succeeds if and only if *ToTest* is an atom. If *ToTest* is uninstantiated, *current_atom*/1 backtracks through all known atoms. If the argument is not an atom or an unbound variable, *current_atom*/1 fails.

See Also

current_predicate/1
current_predicate/2
current_op/3
def/3
defs/2
pdict/3
predicate_property/2

current_op/3

get the name, type and precedence of a currently defined operator

current_op(*Precedence*, *Type*, *Name*)

?Precedence	<integer> or <variable>
?Type	<atom> or <variable>
?Name	<atom> or <variable>

Succeeds when the atom *Name* is an operator of type *Type* and precedence *Precedence*. Any of the arguments may be uninstantiated variables. To declare operators use the *op*/3 predicate.

See Also

current_atom/1
current_predicate/1
current_predicate/2
def/3
defs/2
pdict/3
predicate_property/2

current_predicate/1

check or get a current predicate

current_predicate(*Pred*)

?Pred	<pred_spec> or <variable>

Returns the name/arity of a currently defined user predicate and backtracks to find alternative solutions.

See Also

current_atom/1
current_predicate/2
current_op/3
def/3
defs/2
pdict/3
predicate_property/2

current_predicate/2

check or get a current predicate

current_predicate(*Name*, *Term*)

?Name	<atom> or <variable>
?Term	<term> or <variable>

Unifies *Name* with the name of a user defined predicate, and *Term* with the most general term corresponding to that predicate.

See Also

current_atom/1
current_predicate/1
current_op/3
def/3
defs/2
pdict/3
predicate_property/2

date/3

system date

date(*Day*, *Month*, *Year*)

-Day	<variable>
-Month	<variable>
-Year	<variable>

Return the system date in *Day*, *Month* and *Year* as integers.

See Also

date/4
ms/2
ticks/1
time/4

date/4

date to day number conversion

date(*Day*, *Month*, *Year*, *Index*)

?Day	<variable> or <integer>
?Month	<variable> or <integer>
?Year	<variable> or <integer>
?Index	<variable> or <integer>

If the *Day*, *Month* and *Year* are given as positive integers then *Index* returns the number of days since January 1st 1600. If the *Index* is given as a positive integer representing the number of days since January 1st 1600, then *Day*, *Month* and *Year* return the date specified.

Note: The algorithm takes into account all rules concerning leap years according to the Gregorian calendar.

See Also

date/3
ms/2
ticks/1
time/4

dde_advise_dict/1

get or check the list of open advise loops

dde_advise_dict(*Dict*)

?Dict <list_of <term> > or <variable>

Returns or checks a given list of open advise loops. *Dict* is a list of terms of the form:

(ChannelName, AdviseLoopName)

Where *ChannelName* is the name of an open channel to source opened using *dde_open/3*, and *AdviseLoopName* is the name of an advise loop associated with *ChannelName* opened using *dde_open_advise/4* .

See Also

dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_channel_dict/1

get or check a list of open source channels

dde_channel_dict(*Dict*)

?Dict <list_of <atom> > or <variable>

Each member of *Dict* is the logical name of an open source channel that was previously opened using *dde_open/3*.

See Also

dde_advise_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_close/1

close a source channel

dde_close(*ChannelName*)

+ChannelName <atom>

Closes an open source channel and any associated advise loops. *ChannelName* is the logical name assigned by the user when the channel was opened using *dde_open/3*.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_close_advise/2

close an advise loop for a given open channel

dde_close_advise(*Channel, AdviseName*)

+Channel <atom>

+AdviseName <atom>

The advise loop *AdviseName* for the channel *Channel* is closed. Where *Channel* is the logical name given by the user of the channel opened by *dde_open/3* and *AdviseName* is the logical name given by the user of an advise loop opened by *dde_open_advise/4*.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_close_all/0

close all source channels

Closes all open source channels and associated advise loops. This does not affect any registered topics.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_close_all_topics/0

close all registered topics

Closes all registered topics. Topics are registered using the predicate *dde_open_topic/2*.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_close_topic/1

close a named topic

dde_close_topic(*Topic*)

+Topic <atom>

Closes a registered topic. *Topic* is the logical name given to the topic by the user when it is registered using *dde_open_topic/2*.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_dll_name/1

gets or checks the absolute file name of the DDE Dynamic Link Library

dde_dll_name(*File*)

?File <atom>

This predicate returns the file name and location of the DDE.DLL file. Normally this is located in the SYSTEM sub-directory of the Prolog home directory, though this location may be overrided using *dde_dll_file_name/1*. The predicate *dde_dll_name/1* is used by *dde_load/0* to find the DDE DLL file.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_dll_file_name/1

user-defined fact for setting the absolute file name of the DDE Dynamic Link Library

dde_dll_file_name(*File*)

?File <atom>

This hook is provided to allow the location of the DDE.DLL to be specified in stand-alone applications. The File argument denotes the full path name of the DDE.DLL including the file name and extension. This fact, if present, is used by *dde_dll_name/1* to return the location of the DLL at runtime.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_enable_state/2

Get or set the enable state of a channel

dde_enable_state(*Channel*, *State*)

+Channel	<atom>
+Execute	<atom> in the domain {on,off} or <variable>

If *Execute* is an atom in the range { on, off }, then, *off* will disable the channel and *on*, will enable it. disabling a channel stops all transactions on that channel. It is the equivalent of an interrupt flag.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_execute/2

start an execute transaction

dde_execute(*Channel, Execute*)

+Channel	<atom>
+Execute	<string> or <atom>

An execute transaction is requested for the channel *Channel*. The *Execute* variable should conform to the source's expectations of an executable string. Channels can be opened using *dde_open/3*.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_fetch_data/1

fetch data for a transaction

dde_fetch_data(*Data*)

-Data <variable>

The predicate *dde_fetch_data/1* is used to fetch data from a channel after a DDE transaction in which data was passed.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_load/0

load the DDE Dynamic Link Library

Loads the DDE Dynamic Link Library, this has to be performed before any other DDE functions can be used correctly.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_open/3

open a DDE source channel

dde_open(*ChannelName, Service, Topic*)

+ChannelName	<atom>
+Service	<atom>
+Topic	<atom>

Open a channel with the logical channel name (*ChannelName*) for the given application (or service) *Service* on the *Topic*. If the channel is a previously opened channel, then that channel is closed before the new channel is opened.

The service name *Service* is typically the name of the source application, minus the extension. Thus the service name of *FOO.EXE* say could be *'FOO'* or *foo.* If the named service is not currently running you can run it by using *exec/3*.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_open_advise/4

open an advise loop

dde_open_advise(*AdviseName, Channel, Item, Handler*)

+AdviseName	<atom>
+Channel	<atom>
+Item	<string> or <atom>
+Handler	<atom>

Open an advise loop *AdviseName* on the open source *Channel* and the given *Item* and register the *Handler* to it. An advise loop is a loop where the destination (Prolog) is notified of any changes to specified data.by the source.

The handler is a user-defined Prolog program of the form:

```
Handler( Channel, Data ) :- ...
```

where *Handler* is the functor of the program passed as the fourth argument of *dde_open_advise/4*, *Channel* is the logical name of the source channel and *Data* will be bound to the changed data from that advise loop on receipt of the message.

On receipt of an advise loop message the handler should first set the channel enable state to off, using *dde_enable_state/2*, then process the changed data and finally reset the channel enable state back to on again using *dde_enable_state/2*.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_open_topic/2

open a topic

dde_open_topic(*Topic, Handler*)

+Topic	<atom>
+Handler	<atom>

Opens a DDE *Topic* with the specified *Handler*.

The handler is a Prolog program of the form:

```
Handler( Transaction, Item ):- ...
```

where *Handler* is the functor of the program passed as the second argument of *dde_open_topic/2*, *Transaction* is in the domain { request, poke, execute } and *Item* is the item for the topic on which this transaction is requested.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_poke/3

poke data to a channel

dde_poke(*Channel, Item, Data*)

+Channel	<atom>
+Item	<atom>
+Data	<atom> or <string>

Poke data to a specified item within a given channel. The *Channel* argument is an atom that specifies a previously opened channel. The *Item* argument is an atom that specifies a particular item within the specified channel. The *Data* argument is an atom or a string that is passed to the channel.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_put_data/1

put data to a channel

dde_put_data(*Data*)

+Data <term>

The predicate *dde_put_data/1* is used to put data to a conversation in response to a request transaction.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_request/3
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_request/3

a DDE request transaction.

dde_request(*Channel, Item, Value*)

+Channel	<atom>
+Item	<atom>
-Value	<variable>

Request a value from the given channel and item. The *Value* argument is returned as a string.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_timeout/1
dde_topic_dict/1
dde_unload/0

dde_timeout/1

get or set the DDE time out value

dde_timeout(*Time*)

?Time	<integer> or <variable>

Gets or sets the DDE time out value, the value is in milliseconds. This is a global value affecting all source channels, the default is 5000ms. A transaction will generate a timeout error if the source has not responded within the given timeout period.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_topic_dict/1
dde_unload/0

dde_topic_dict/1

get or check a list of open topics

dde_topic_dict(*Dict*)

?Dict	<list_of <atom> >or <variable>

Get or check a list of registered topics. Topics can be registered using *dde_open_topic/2*.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_unload/0

dde_unload/0

unload the DDE Dynamic Link Library

Unloads the DDE Dynamic Link Library and closes all channels, associated advise loops and topics.

See Also

dde_advise_dict/1
dde_channel_dict/1
dde_close/1
dde_close_advise/2
dde_close_all/0
dde_close_all_topics/0
dde_close_topic/1
dde_dll_name/1
dde_enable_state/2
dde_execute/2
dde_fetch_data/1
dde_load/0
dde_open/3
dde_open_advise/4
dde_open_topic/2
dde_poke/3
dde_put_data/1
dde_request/3
dde_timeout/1
dde_topic_dict/1

'?DEBUG?'/1

user-defined Prolog program which intercepts calls to the debugger

'?DEBUG?'(*Goal*)

+Goal <goal>

User-defined program to intercept Prolog debugger calls. The *Goal* argument is the Prolog goal that was about to be sent to the debugger. The debug hook is invoked whenever a spied predicate is called and debugging mode is set to "debug". To allow the default processing of debugging calls you should call *debug_hook/1*.

See Also
debug_hook/1

debug/0

set the debug mode to on

Sets the debug mode to be *on*. The debugger will be invoked at the next spied predicate that is called.

See Also

debug_hook/1
debugging/0
force/1
leash/2
leashed/2
ms/2
no_style_check/0
no_style_check/1
nodebug/0
nospy/1
nospyall/0
notrace/0
spy/1
style_check/0
style_check/1
trace/0

debug_hook/1

system handler for the debug hook

debug_hook(*Goal*)

+Goal <term>

Invoke the currently set system debugger with the given *Goal*. The predicate *debug_hook/1* is mainly provided to allow programmatic access to the system debugger in user-defined *'?DEBUG?'/1* programs.

See Also

debug/0
debugging/0
force/1
leash/2
leashed/2
ms/2
no_style_check/0
no_style_check/1
nodebug/0
nospy/1
nospyall/0
notrace/0
spy/1
style_check/0
style_check/1
trace/0

debugging/0

write the current status of the debugger to the standard output stream

Writes the current status of the debugger to the standard output stream

It writes the following:

The debugging mode (debug, trace or off).

The leash ports.

All of the predicates that have spy points set.

See Also

debug/0
debug_hook/1
force/1
leash/2
leashed/2
ms/2
no_style_check/0
no_style_check/1
nodebug/0
nospy/1
nospyall/0
notrace/0
spy/1
style_check/0
style_check/1
trace/0

def/3

check for a currently defined predicate and return its type

def(*Functor*, *Arity*, *Type*)

+Functor	<functor>
+Arity	<arity>
?Type	<variable> or <integer> in the domain [0,1,2,3,4]

Get or check the type of a predicate. The *Functor* argument is an atom that is the functor of the predicate whose type is required. The *Arity* argument is an integer that is the number of arguments of the predicate whose type is required. The *Type* argument is either a variable or a predicate type value. Predicates are divided by *def*/3 into five main categories: null-predicates, dynamic predicates, static and optimised predicates, internal assembler code predicates and external dynamically linked assembler code predicates.

Programs that need to differentiate between predicate types can be written using *def*/3 combined with efficient first argument indexing to switch to some appropriate code.

See Also

current_atom/1
current_predicate/1
current_predicate/2
current_op/3
defs/2
pdict/3
predicate_property/2

defs/2

return all arities for a given functor

defs(*Functor*, *Arities*)

+Functor	<functor>
-Arities	<variable>

Return a list of defined *Arities* for the given *Functor*.

See Also

current_atom/1
current_predicate/1
current_predicate/2
current_op/3
def/3
pdict/3
predicate_property/2

del/1

delete a file

del(*File*)

+File <atom>

Deletes the disk file whose name is *File* (*File* must be an atom). *del*/1 does not add a default file extension to *File*.

It is possible to delete files that are currently open for reading and writing. However, this is not recommended as it tends to result in the disk becoming clogged with file fragments. If you do delete an open file, then use the DOS command:

```
chkdsk /f
```

to recover the files and disk space.

See Also

absolute_file_name/2
absolute_file_name/3
cat/3
close/1
fclose/1
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
attrib/2
chdir/1
dir/3
drive/1
env/2
fcreate/3
ren/2
rmdir/1
stamp/1

dict/1

return a dictionary of atoms

dict(*Atoms*)

-Atoms <variable>

Return a dictionary of all the non-hidden atoms as a list.

See Also
current_atom/1
current_predicate/1
current_predicate/2
current_op/3
def/3
defs/2
pdict/3
predicate_property/2

dir/3

get a file directory

dir(*Patt*, *Attrib*, *Files*)

+Patt	<atom>
+Attrib	<integer>
-Files	<variable>

The *Patt* argument specifies a file name pattern to be searched for. The *Attrib* argument is a file type value that filters the files that match the pattern. The *Files* argument is the final list of filtered files.

These file type values are additive, so if you wanted to return all the read/write files and all the directories you could enter the following query:

```
?- dir(*.*, 17, Filelist).
```

If the value for the attributes is a positive integer then the returned list consists of entries of the form:

```
[filename, [day,month,year,hours,mins,secs], size, attribute]
```

If the value for the attributes is a negative integer then the returned list consists of filenames only.

See Also

absolute_file_name/2
absolute_file_name/3
cat/3
close/1
fclose/1
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
attrib/2
chdir/1
del/1
drive/1
env/2
fcreate/3
ren/2
rmdir/1
stamp/1

dirbox/4

display the directory box

dirbox(*Title*, *Message*, *Pattern*, *Pathname*)

+Title	<string>
+Message	<atom> or <string>
+Pattern	<atom>
-Pathname	<variable>

Displays the "directory" dialog box with the given title (window caption) and message. This dialog includes two list boxes, one for filenames and one for pathnames, and an "edit" field for entering a file name. The initial contents of the "edit" field and list boxes is computed from the given file pattern.

The predicate succeeds if the OK button is clicked or RETURN is pressed and a file name is selected, or fails if the CANCEL button is clicked or ESCAPE is pressed. If OK or RETURN occur while a file pattern (a name containing one or more wildcard characters) is selected, the list boxes are updated with file matching the new pattern, and the dialog continues. Upon success, it returns the full path name of the selected file.

See Also

abtbox/3
change_hook/3
chgbox/3
erase_status_box/0
find_hook/3
fndbox/2
message_box/3
msgbox/4
status_box/1
sttbox/2

display/1

write a term to the standard output stream in standard prefix notation

display(*Term*)

?Term	<term>

The predicate *display/1* writes all compound terms in the standard prefix notation (ignoring the current operator declarations).

Like *write/1*, *display/1* does not output a dot after the term. If the term is to be read back in, the terminating dot and space character must be output explicitly. Unlike *write/1*, any control characters in *Term* are output literally.

See Also

current_op/3
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
read/1
sysops/0
skip_term/0
vars/2
write/1
write_canonical/1
writeq/1
prompt/2

'?DLL?'/3

user-defined hook for handling DLL messages

'?DLL?'(*Message*,*Data*,*Goal*)

+Message	<integer>
+Data	<integer>
+Goal	<goal>

The *'?DLL?'/3* program is used to handle an interrupt generated by a DLL message. The *Message* argument will be matched with the message number that was sent. The *Data* argument will be matched with any data associated with the message. The *Goal* argument is the Prolog goal that was interrupted by the message. To call the default behaviour for handling DLLs you should call the predicate *dll_hook/3*.

See Also

dll_hook/3
lcall/4
lclose/1
ldict/1
lopen/1
'?DLL?'/3
winapi/5

dll_hook/3

system defined handling for the dll hook

dll_hook(*Message, Data, Goal*)

+Message	<integer>
+Data	<integer>
+Goal	<term>

Invoke the system defined behaviour for the given DLL *Message*, *Data* and then run the interrupted *Goal*. The predicate *dll_hook/3* displays a response to show that the given DLL *Message* and *Data* were not handled by the system.

The predicate *dll_hook/3* is mainly provided to allow programmatic access to the system debugger in user-defined *'?DLL?'/3* programs.

See Also

lcall/4
lclose/1
ldict/1
lopen/1
'?DLL?'/3
winapi/5

dos/0

initiate a DOS shell

The predicate *dos*/0 initiates a DOS shell from within Prolog. Once inside the shell you can perform any DOS command, when you need to return to Prolog type EXIT on the DOS command line to exit the shell.

See Also
dos/1
exec/3
switch/2
ver/4

dos/1

initiate a DOS shell and run the given command

dos(*Program*)

+Program <atom>

Initiate a DOS shell and then execute a program. The *Program* argument must be an atom that is the name of the program to be executed.

See Also
dos/0
exec/3
switch/2
ver/4

drive/1

choose or return a drive

drive(*Drive*)

?Drive	<atom> or <variable>

Finds or changes the current disk drive. The current drive is where Prolog searches for a file if a drive name is not explicitly specified.

If *Drive* is a variable *drive*/1 will bind it to the name of the current drive. This drive name will be a single upper case character.

If *Drive* is a single character atom which designates one of the disk drives, the named drive will become the current drive. *Drive* can be an upper or lower case character. If *Drive* does not exist, the current drive is left unaltered. *drive*/1 will fail if *Drive* is not a single character.

See Also

absolute_file_name/2
absolute_file_name/3
cat/3
close/1
fclose/1
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
attrib/2
chdir/1
del/1
dir/3
env/2
fcreate/3
ren/2
rmdir/1
stamp/1

dynamic/1

define a predicate to be dynamic

dynamic *Pred*

+Pred <pred_specs>

A procedure is declared dynamic by preceding its definition with a declaration of the form:

```
:- dynamic Pred
```

where *Pred* must be a procedure specification of the form Name/Arity, or a sequence of such specifications, separated by commas. Any existing clauses for the predicate will be removed.

There are two types of dynamic predicate.

The first is 'normal' dynamic: when a predicate is called its actual code is utilized, with the result that it is not safe to make modifications to this code during its execution. It is, of course, perfectly safe to modify any code which is not being executed at the time. This type of dynamic code is faster than 'logical' dynamic code.

The second is 'logical' dynamic: when a predicate is called, a copy is made of all the clauses matching the goal, and this is executed as a meta-call. The code itself is not executed, so changes can safely be made at any time.

The type of dynamic predicates used is set globally and can be specified when you run Prolog by using the /D switch. The following command will run Prolog using 'logical' dynamic predicates:

```
C> PRO386 /D1
```

The default for the /D switch is 0 and this corresponds to the use of 'normal' dynamic predicates.

See Also

abolish/1
abolish/2
abolish_files/1
assert/1
asserta/1
assert/2
assertz/1
clause/2
clauses/2
clause/3
functor/3
listing/0
listing/1
retract/1
retractall/1
retract/2
volatile/1

dynamic_call/1

call a dynamic procedure safely

dynamic_call(*Goal*)

+Goal <goal>

Ensures that the call to the dynamic *Goal* will not be affected by subsequent calls to assert and retract predicates during its evaluation.

Asserts and retracts are always done immediately. This means that calls of the form:

```
?- foo(X), retract( foo(Y) ).
```

will not behave logically, since the retraction will side-affect the database immediately and so be visible to the initial call to foo/1. This can be averted by calling foo/1 in the following manner:

```
?- dynamic_call( foo( X ) ), retract( foo(Y) ).
```

N.B. The safe *logical* treatment of dynamic predicates can be done automatically when Prolog is strarted by including the switch: /D.

See Also

abolish/1
abolish/2
abolish_files/1
assert/1
asserta/1
assert/2
assertz/1
clause/2
clauses/2
clause/3
dynamic/1
functor/3
listing/0
listing/1
retract/1
retractall/1
retract/2
volatile/1

elex/1

set, reset or get the edinburgh flag

elex(*Flag*)

?Flag	<variable> or <integer> in the domain {0,1}

Set or get the Edinburgh language extensions *Flag*. Prolog has certain built-in extensions to the standard Edinburgh syntax, one of these being the use of the backwards quote symbol to denote Prolog string data types. These extensions generally should not affect standard Edinburgh syntax programs: if, however, problems are encountered due to these extensions, Prolog provides the option of turning them off, using the *elex*/1 predicate. To turn the syntax extensions off enter the following goal:

```
?- elex( 0 ).
```

To turn the syntax extensions on use the following goal:

```
?- elex( 1 ).
```

Note: by default the syntax extensions are on.

See Also

<~/2
~>/2
cat/3

ensure_loaded/1

load the specified Prolog source and/or object file(s) into memory.

ensure_loaded(*FileSpec*)

+FileSpec <file_specs>

ensure_loaded/1 loads each of the specified files except for files which have previously been loaded and which have not been changed since they were last loaded.

FileSpec can be a file specification of the form: *PathAlias*(*File*) where *PathAlias* is an alias that refers to a specified path and *File* is a file name relative to that path. *FileSpec* can also be an atom of the form: *Path/File* where *Path* defines the path where the file *File* will be found.

Note: a ".PL" or ".PC" extension may be omitted in a file specification.

See Also

abolish_files/1
compile/1
consult/1
initialization/1
load_files/1
load_files/2
multifile/1
prolog_load_context/2
reconsult/1
save_predicates/2
source_file/1
source_file/2
source_file/3

env/2

get an environment string

env(*Name*, *List*)

+Name	<atom>
?List	<variable>

Succeeds if *List* is the value of the DOS environment variable *Name*. Environment variables are initialised from DOS using the SET command (see your DOS documentation for details). The use of environment variables provides a simple method of communication between Prolog and other applications.

Name must be an atom that is the name of the DOS environment variable. It can be in upper or lower case. *env*/2 will fail if it does not refer to the name of an environment variable.

Note: if *Name* is the empty atom (i.e. ''), then the name of the Prolog kernel together with its home directory is returned in *List*.

List must be an unbound variable. It will be bound to the list of alternative values of *Name*. DOS uses the semicolon character (';') to denote alternative values.

See Also

dos/0
dos/1
exec/3
switch/2
ver/4

eprint/1

print a quoted edinburgh term to the current output stream

eprint(*Term*)

+Term <term>

Write *Term* to the current output stream. Single quotes are put around any atoms in *Term* that would have to be quoted on input. *eprint*/1 does not terminate the term with a full stop, so if you are writing to a file, and want to read the term back again you must follow the *eprint*/1 with either of the following:

```
write(' . ').
write(' .'),nl.
```

See Also

current_op/3
display/1
elex/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
read/1
sysops/0
skip_term/0
vars/2
write/1
write_canonical/1
writeq/1
prompt/2

eprint/2

same as eprint/1 but with the ability to output variable names

eprint(*Term*, *Vars*)

+Term	<term>
+Vars	<list_of (<atom>,<variable>) >

Write a *Term* in quoted Edinburgh syntax to the current output stream using variable names instead of variables, where *Vars* defines the association between the names and the variables in the *Term*. *Vars* should be a list of (*Name*,*Variable*) comma pairs. Each comma pair specifies that any occurrence of the *Variable* in the *Term* should be replaced with *Name*.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
read/1
sysops/0
skip_term/0
vars/2
write/1
write_canonical/1
writeq/1
prompt/2

eprint/3

same as eprint/2 but with added precedence

eprint(*Term*, *Vars*, *Precedence*)

+Term	<term>
+Vars	<list_of (<atom>,<variable>) >
+Precedence	<integer> ≥ -1

Write a *Term* in quoted Edinburgh syntax to the current output stream using variable names instead of variables, where *Vars* defines the association between the names and the variables in the *Term*. *Vars* should be a list of comma pairs of the form: (*Name*,*Variable*). Each comma pair specifies that any occurrence of the *Variable* in the *Term* should be replaced with *Name*.

The *Precedence* specifies whether the output should be bracketed. If the precedence of any operators in the term to be output is higher than the given *Precedence* then the term is output with brackets. This method of printing terms can be useful when recursively stepping through and printing Prolog structures that may contain operators.

If the *Priority* is a positive integer or zero then the output mode is infix. If instead the *Priority* is -1 then the output mode is prefix.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
read/1
sysops/0
skip_term/0
vars/2
write/1
write_canonical/1
writeq/1
prompt/2

eqv/2

check two terms for equivalence

eqv(*First*, *Second*)

+First	<term>
+Second	<term>

Check the equivalence of two terms *First* and *Second*. When comparing the equality of the terms a check is made to see if an uninstantiated variable in a particular argument position in either term corresponds to an uninstantiated variable in the *same* position in the other term. Note: the variables do not need to be identical.

See Also

=/2
==/2
@</2
@=</2
@>/2
@>=/2
\=/2
\==/2
cmp/3
compare/3
keysort/2
len/2
sort/2
sort/3
occurs_chk/2
subsumes_chk/2

erase_status_box/0

destroy the status message window

Closes the status box window explicitly. This predicate is recommended for its portability as it is available on both the Windows and MS-DOS platforms it is equivalent to calling *sttbox/2* with the second argument set to -1.

See Also
status_box/1
message_box/3

eread/1

read an edinburgh term from the current input stream

eread(*Term*)

-Term <variable>

Read the next *Term* from the current input stream. In the input stream, the term must be followed by a dot ('.') and at least one white space character (i.e. a character whose ASCII code is less than or equal to 32). The dot and white space character are read in but are not considered part of the term.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
read/1
sysops/0
skip_term/0
vars/2
write/1
write_canonical/1
writeq/1
prompt/2

eread/2

same as eread/1 but with added var list

eread(*Term*, *Vars*)

-*Term* <variable>

-*Vars* <variable>

Read the next *Term* from the current input stream and return the associated variable names and variables in the *Term*. *Vars* will be instantiated to a list of comma pairs of the form: (*Name*,*Variable*), where *Variable* is a variable that occurs in the *Term* and *Name* is the name used to represent that variable.

The predicate *eread*/2 can be used in conjunction with *eprint*/2 and *ewrite*/2 to maintain meaningful variable names.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
read/1
sysops/0
skip_term/0
vars/2
write/1
write_canonical/1
writeq/1
prompt/2

'?ERROR?'/2

user-defined Prolog program which intercepts error messages

'?ERROR?'(*Number*, *Goal*)

+Number	<integer>
+Goal	<goal>

User-defined program to intercept Prolog error messages. The *Goal* argument is the goal that threw the error. The error hook is called whenever an error is thrown to the system (for more information on the reporting of errors see *catch/2* and *throw/2*). There is an error which is not actually passed to the error hook and this is *error 11, keyboard break* (triggered by pressing the <ctrl-break> key) which invokes the break hook instead (see *'?BREAK?'/1*). The *Number* argument is the number of the error that was thrown.

To allow the default processing of error messages you should call *error_hook/2*.

Please note that error numbers 0, 1, 2, 3, 4, 7 and 8 are handled at a lower-level than most errors. This is because if a Prolog program was allowed to handle them, as is the case for the other errors, this could itself re-generate the original error giving rise to the situation where an infinite error loop occurs.

See Also
error_hook/2

error_hook/2

system defined behaviour for error handling

error_hook(*Error, Goal*)

+Error	<integer>
+Goal	<term>

Invoke the system defined error handler for the given numbered *Error* and the given *Goal*. The predicate *error_hook/2* displays an error message that gives the number of the error, the text of what the error number represents and the goal that generated the error and then aborts the goal returning control to the Prolog supervisor. The predicate *error_hook/2* is mainly provided to allow programmatic access to the default system error handler in user-defined *'?ERROR?'/2* programs.

Please note that error numbers 0, 1, 2, 3, 4, 7 and 8 are handled at a lower-level than most errors. This is because if a Prolog program was allowed to handle them, as is the case for the other errors, this could itself re-generate the original error giving rise to the situation where an infinite error loop occurs.

See Also

abort/0
error_message/2
unknown_predicate_handler/2
catch/2
catch/3
throw/2
flush/0

error_message/2

return an error message for an error number

error_message(*Number*, *Message*)

+Number	<integer>
-Message	<variable>

The *error_message*/2 predicate can be used to find the error message associated with an error number.

See Also

abort/0
unknown_predicate_handler/2
catch/2
catch/3
throw/2
error_hook/2
flush/0

etoks/1

read an edinburgh token list from the current input stream

etoks(*TokenList*)

-TokenList <variable>

Read a list of tokens from the current input stream to be returned in the variable *TokenList*. Each element of *TokenList* will be a conjunction of the form: (*Type*, *Token*) - where Token is one token from the input stream and *Type* is a token value that shows the data type of the token.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
read/1
sysops/0
skip_term/0
vars/2
write/1
write_canonical/1
writeq/1
prompt/2

etoks/2

read an edinburgh token list from the current input stream with variable names

etoks(*TokenList, VarNames*)

-*TokenList* <variable>

-*VarNames* <variable>

Read a list of tokens from the current input stream and return the association between any variables and their variable names. *TokenList* is the list of tokens returned. Each element of *TokenList* will be a conjunction of the form: (*Type*,*Token*) - where *Token* is one token from the input stream and *Type* is a number denoting the data type of the token according to table X. *VarNames* is a list of the associations between any variables in *TokenList* and their variable names. Each element of *VarNames* will be a conjunction of the form: (*VarName*,*Var*) - where *VarName* is the quoted name of the variable and *Var* is the internal representation of the variable.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
read/1
sysops/0
skip_term/0
vars/2
write/1
write_canonical/1
writeq/1
prompt/2

ewrite/1

write an unquoted edinburgh term to the current output stream

ewrite(*Term*)

+Term	<term>

Write *Term* to the current output stream. All quoted atoms in *Term* are output unquoted. *ewrite*/1 does not terminate the term with a full stop, so if you are writing a term to a file, and want to read that term back again you must follow the *ewrite*/1 with either of the following:

```
write(' . ').
write(' .'),nl.
```

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
read/1
sysops/0
skip_term/0
vars/2
write/1
write_canonical/1
writeq/1
prompt/2

ewrite/2

same as ewrite/1 but with the ability to output variable names

ewrite(*Term*, *Vars*)

+Term	<term>
+Vars	<list_of (<atom>,<variable>) >

Write a *Term* in unquoted Edinburgh syntax to the current output stream using variable names instead of variables, where *Vars* defines the association between the names and the variables in the *Term*. *Vars* should be a list of (*Name*,*Variable*) comma pairs. Each comma pair specifies that any occurrence of the *Variable* in the *Term* should be replaced with *Name*.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/3
op/3
portray_clause/1
print/1
printq/1
read/1
sysops/0
skip_term/0
vars/2
write/1
write_canonical/1
writeq/1
prompt/2

ewrite/3

same as ewrite/2 but with added priority

ewrite(*Term*, *Vars*, *Mode*)

+Term	<term>
+Vars	<list_of (<atom>,<variable>) >
+Mode	<integer> -1

Write a *Term* in unquoted Edinburgh syntax to the current output stream using variable names instead of variables, where *Vars* defines the association between the names and the variables in the *Term*. *Vars* should be a list of comma pairs of the form: (*Name*,*Variable*). Each comma pair specifies that any occurrence of the *Variable* in the *Term* should be replaced with *Name*.

The *Mode* specifies whether the output should be in infix or prefix mode. If the *Mode* is a positive integer or zero then the output mode is infix. If instead the *Mode* is -1 then the output mode is prefix.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
op/3
portray_clause/1
print/1
printq/1
read/1
sysops/0
skip_term/0
vars/2
write/1
write_canonical/1
writeq/1
prompt/2

exec/3

execute an external program

exec(*Program*, *Args*, *Status*)

+Program	<atom>
+Args	<atom>
-Status	<variable>

Execute an external program with the given arguments. The *Program* argument is the name of the program to be executed given as a quoted atom. It must include the file extension ('.COM' or '.EXE'). The *Args* argument should be a single quoted atom of all the arguments to be passed to the program. The *Status* argument is a variable that is bound to the exit status code of the program.

To run the DOS command processor as a sub-process *exec/*3 can be used in the following way:

```
exec(command, '', S).
```

where *command* is the full path name of the command processor (COMMAND.COM). This name is given by the environment variable COMSPEC. The following query will execute the command processor.

```
env(comspec, C), C = [Cmd], exec(Cmd, '', X).
```

To return to Prolog from the DOS command processor type the command:

```
A>exit
```

To run DOS commands *exec/3* can be used in the following way:

```
exec(command, '/C action', S).
```

where *command* is the full name of the command processor (see above), and *action* is the DOS command to be executed. This mechanism can be used for running both internal and external DOS commands.

To run external DOS commands and user defined programs *exec/3* can be used in the following way:

```
exec(program, args, S).
```

where *program* is the full name of the program to run (including file extension) and *args* is the list of arguments.

The advantage of invoking a program directly is that it avoids the overhead of loading the command processor (COMMAND.COM). The disadvantage is that you must specify the full path name of the program - *exec/3* does not perform a path search to find it.

See Also

dos/0
dos/1
env/2
switch/2
ver/4

expand_dcg/2

convert grammar rules to Prolog without calling term_expansion/2

expand_dcg(*T1*, *T2*)

+T1	<term>
-T2	<variable>

If *T1* is a grammar rule then *T2* is the corresponding Prolog representation of that rule. Otherwise *T2* is identical to *T1*.

The predicate *expand_dcg/2* converts grammar rules to Prolog without attempting to call the user-defined *term_expansion/2*. This predicate is provided so that users wanting to do their own term expansion can additionally perform the default grammar rule expansion.

See Also

'C'/3
expand_term/2
phrase/2
phrase/3
term_expansion/2

expand_term/2

convert between a grammar rule and its Prolog equivalent

expand_term(*T1*, *T2*)

+T1	<term>
-T2	<variable>

If *T1* is a grammar rule then *T2* is the corresponding Prolog representation of that rule. Otherwise *T2* is identical to *T1*. This program also tries the user-defined *term_expansion*/2. The predicate *expand_term*/2 is automatically called when compiling and consulting programs.

See Also

'C'/3
expand_dcg/2
phrase/2
phrase/3
term_expansion/2

fail/0

force failure

Always fails. Can be used to force backtracking in a query.

See Also
!/0
./2
->/2
:/2
\+/1
abort/0
false/0
halt/0
halt/1
not/1
otherwise/0
repeat/0
repeat/1
true/0
break/0
break_hook/1

false/0

force failure

Always fails. Synonym for fail.

See Also
!/0
./2
->/2
:/2
\+/1
abort/0
fail/0
halt/0
halt/1
not/1
otherwise/0
repeat/0
repeat/1
true/0
break/0
break_hook/1

fclose/1

close a file

fclose(*File*).

+File <atom>

The predicate *fclose*/1 closes the named file, the file buffer is written to disk and the file is closed. This predicate can only close files if it uses the identical file name that is found on the file dictionary. The file dictionary can be found using the predicate *fdict*/1.

See Also

absolute_file_name/2
absolute_file_name/3
cat/3
close/1
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
attrib/2
chdir/1
del/1
dir/3
drive/1
env/2
fcreate/3
ren/2
rmdir/1
stamp/1

fcreate/3

create a file with the given attributes

fcreate(*Logicalname*, *Filename*, *Attrib*).

+Logicalname	<atom>
+Filename	<atom>
+Attrib	<integer> in the domain {0,1,2,3}

The predicate *fcreate*/3 creates a file with certain access attributes. The *Logicalname* argument gives the name that Prolog will use to subsequently access the file. The *Filename* argument gives the name of the file as it will appear on the disk. The *Attrib* argument is a file attribute value that gives the access attributes for the file. Initially any file created using *fcreate*/3 is automatically opened with read/write access. After closing the file it then gains the access attributes given in the call to *fcreate/*3.

Note: if the filename that is given to *fcreate*/3 already exists, *no* attempt will be made to create a back-up file. This has been done to allow users to develop their own systems for backing up files.

See Also

absolute_file_name/2
absolute_file_name/3
cat/3
close/1
fclose/1
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
attrib/2
chdir/1
del/1
dir/3
drive/1
env/2
ren/2
rmdir/1
stamp/1

fdict/1

return a dictionary of files

fdict(*Dict*)

-Dict <variable>

Return a list of all the currently open files. You can use this predicate to check the actual representation used internally to reference currently open files.

See Also

absolute_file_name/2
absolute_file_name/3
cat/3
close/1
fclose/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
attrib/2
chdir/1
del/1
dir/3
drive/1
env/2
fcreate/3
ren/2
rmdir/1
stamp/1

file_search_path/2

user defined fact specifying a path name

file_search_path(*Dirspec, Directory*)

+Dirspec	<path_alias>
+Directory	<file_spec>

Dirspec is a logical name that will be used for specifying a directory, and *Directory* is the path of the directory to be aliased in this way.

See Also

absolute_file_name/2
absolute_file_name/3
cat/3
close/1
fclose/1
fdict/1
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
attrib/2
chdir/1
del/1
dir/3
drive/1
env/2
fcreate/3
ren/2
rmdir/1
stamp/1

fileerrors/0

turn on the reporting of file error messages

Sets the 'fileerrors' flag to its default state in which an error message is reported by *see*/1, *tell*/1 and *open*/2 if the specified file cannot be opened. The error message is followed by an *abort*/0, execution is abandoned and the system returns to top level.

The 'fileerrors' flag is only disabled by an explicit call to *nofileerrors*/0, or via *prolog_flag*/3 which can also be used to obtain the current value of the 'fileerrors' flag.

See Also

no_style_check/0
no_style_check/1
nofileerrors/0
prolog_flag/2
prolog_flag/3
style_check/0
style_check/1
prompt/2
prompts/2
switch/2

find/1

find a string or atom in a file

find(*ToFind*)

+ToFind <string> or <atom>

The predicate *find*/1 reads characters from current input stream until there is a match with *ToFind*. This leaves the cursor at first character after the matching characters. If *ToFind* is the string: ``, or the atom: '', then search the current input stream for the beginning of the next non-white space text.

See Also

at_end_of_file/0
at_end_of_line/0
inpos/1
outpos/1
skip/1
skip_layout/0
skip_line/0
skip_term/0
stream_position/2
stream_position/3
flush/0

'?FIND?'/3

user-defined hook for handling find box messages

"?FIND?'(*Focus*, *Message*, *Goal*)

+Focus	<window_handle>
+Message	<integer>
+Goal	<goal>

Where *Focus* is the window that is currently in focus, *Message* is the message that was generated and *Goal* is the goal that was interrupted by the *Message*.

In response to one of the find box messages, you should make a call to *fndbox/2* this time giving two variables. The first of these will return the text in the "find" box as a string, and the second will return the radio button choice.

You can invoke the default message handler directly by calling the predicate *find_hook/3*.

See Also
find_hook/3
fndbox/2

find_hook/3

system defined handler for the find dialog

find_hook(*Win, Msg, Goal*)

+Win	<window_handle>
+Msg	<integer>
+Goal	<term>

Invoke the default system behaviour for the find dialog in the given window (*Win*) for the given message (*Msg*) and then run the given *Goal*.

See Also

abtbox/3
chgbox/3
change_hook/3
dirbox/4
erase_status_box/0
fndbox/2
message_box/3
msgbox/4
status_box/1
sttbox/2

findall/3

find all the instances of a term for which a Prolog goal is true

findall(*Term*, *Call*, *List*)

?Term	<term>
+Call	<goal>
?List	<variable>

Succeeds if *List* is the list of all instances of *Term* for which *Call* holds. *Term* may be any type of Prolog term. *Call* must be a goal to be called. *List* will be unified with a list of instantiated copies of *Term*.

Each element of *List* will be a copy of *Term*, and there will be one copy for each different solution to the query. If there are no solutions to the query, *List* will be unified with the empty list.

If *Term* contains a variable that is bound in a solution then the value to which it has been bound will appear in the instantiated copy of *Term*.

The solution list *List* is not sorted. Solutions appear in the list in the same order as they are found.

The solutions in *List* are not necessarily unique. If a different solution to *Call* results in the same value for *Term*, then a duplicate entry will appear in the list.

At the end of an evaluation, any variables in *Term* and *Call* will still be unbound.

See Also

^/2
bagof/3
forall/2
setof/3
solution/2

float/1

test for a floating point number

float(*Term*)

?Term	<term>

Succeeds if *Term* is instantiated to a floating point number. It fails for any other type of term.

See Also

atom/1
atomic/1
callable/1
char/1
chars/1
compound/1
float/1
ground/1
integer/1
integer_bound/3
nonvar/1
number/1
simple/1
string/1
type/2
unifiable/2
var/1

fluff/3

decompress the data in the current input stream to the current output stream

fluff(*BufSize*, *RawCount*, *CompCount*)

-BufSize	<variable>
-RawCount	<variable>
-CompCount	<variable>

This predicate reads compressed data from the current input stream, decompressing them and outputting the raw (uncompressed) equivalent to the current output stream. It terminates when the end of a compressed stream is encountered on input: note, this will not necessarily be the end of file. The window setting that was used for compression (*see stuff/3*) is determined from the input stream and returned in *BufSize*, the total number of raw (uncompressed) bytes processed is returned in *RawCount* and the total number of compressed bytes processed is returned in *CompCount*.

See Also
stuff/3

flush/0

flush the current input stream

Flush the current input stream up to and including the end of the current line of text. *flush*/0 is used to clear keyboard buffer either in error handling or whenever Prolog wants fresh input rather than data that was typed ahead or left over from an earlier input.

See Also

at_end_of_file/0
at_end_of_line/0
find/1
inpos/1
outpos/1
skip/1
skip_layout/0
skip_line/0
skip_term/0
stream_position/2
stream_position/3

fname/4

convert a file name into parts

fname(*FileName*, *Path*, *Name*, *Extension*)

+FileName	<atom>
-Path	<variable>
-Name	<variable>
-Extension	<variable>

Split a given file name into its path, name and extension components. If a path is given, then *Path* will be bound to an atom which always terminates with the '\' character; likewise, if an extension is given the atom returned by *Extension* always begins with a '.' character.

See Also

absolute_file_name/2
absolute_file_name/3
attrib/2
cat/3
chdir/1
close/1
del/1
dir/3
drive/1
env/2
fclose/1
fcreate/3
fdict/1
file_search_path/2
fopen/3
library_directory/1
mkdir/1
open/2
ren/2
rmdir/1
stamp/1

fndbox/2

display the modeless find box

fndbox(*Find*, *Flag*)

?Find	<string> or <variable>
?Flag	<integer> or <variable>

Displays or hides the modeless "find" dialog box with the given prefill find string, or returns data from the existing box. The find dialog includes an "edit" field, labeled "Find:", and three radio buttons, with which the user can select the scope of the find.

The *Find* argument is a string that denotes the contents of the Find: edit field. If an empty string is specified, the existing contents of the find box are not altered. The *Flag* argument is an action value that specifies an action to be performed on the find box.

The predicate succeeds if the find box is created properly, and returns control immediately. When the user later clicks one of the option buttons, the box is disabled and a message sent to Prolog giving details of the user's action.

Normally when the user selects an action in the find box the default message handler will be called. If you want to control the the behaviour of the change box yourself you will need to write a definition for *'?FIND?'/3*.

See Also

abtbox/3
change_hook/3
chgbox/3
dirbox/4
erase_status_box/0
find_hook/3
message_box/3
msgbox/4
status_box/1
sttbox/2
'?FIND?'/3

fonts/1

return a list of available fonts

fonts(*TypeFaces*)

-TypeFaces <variable>

Return a list of all *TypeFaces* avaiable on the current window system. Note, the returned list contains typefaces, not fonts: a font is defined as being a typeface, with a given size and style. The typeface is the raw outline from which you can create fonts (see *wfcreate/4*).

See Also

ansoem/2
wfclose/1
wfcreate/4
wfdata/5
wfdict/1
wfont/2
wfsize/4

fopen/3

open a file with the given access mode

fopen(*Logicalname*, *Filename*, *Mode*).

+Logicalname	<atom>
+Filename	<atom>
+Mode	<integer> in the domain {0,1,2}

The predicate *fopen/3* opens a file with a certain access mode. The *Logicalname* argument gives the name that Prolog will use to subsequently access the file. The *Filename* argument gives the name of the file as it appears on disk. The *Mode* argument is a file access value that gives the access mode for the file.

See Also

absolute_file_name/2
absolute_file_name/3
attrib/2
cat/3
chdir/1
close/1
del/1
dir/3
drive/1
env/2
fclose/1
fcreate/3
fdict/1
file_search_path/2
fname/4
library_directory/1
mkdir/1
open/2
ren/2
rmdir/1
stamp/1

forall/2

generate then test solutions for a goal

forall(*Gen*, *Test*)

+Gen	<goal>
+Test	<goal>

For all the solutions of the generator *Gen*, the *Test* is true. *Gen* and *Test* must be Prolog calls (which may include conjunctions and disjunctions).

See Also

^/2
bagof/3
findall/3
setof/3
solution/2

force/1

call a Prolog goal and suspend the debugger for that call.

force(*Goal*)

+Goal <goal>

Calls the *Goal* and suspends interaction with the debugger. Succeeds if *Goal* succeeds, and fails otherwise. Interaction with the debugger is suspended for that call only: spied sub-calls in the *Goal* will still invoke the debugger. The predicate *force/1* behaves the same as *call/1*, apart from the suspension of the debugger.

Suppose you had the following definition for the program "foo/1":

```
foo(X) :-
        Y is 6 * X,
        write(Y),
        nl,
        bar(X,Y).
```

If there is a spypoint set on the program "foo/1" and "bar/2" and the debugger is set to debug mode, the following goal will force the initial call to "foo/1" to be executed without debugging, but the debugger will be invoked when the goal "bar/2" is called:

```
?- force(foo(3)).
```

This predicate is necessary when you are trying to define the *'?DEBUG?'/1* debugging hook, to stop the spied goal from re-invoking the debugger when it gets called.

See Also

debug/0
debug_hook/1
debugging/0
leash/2
leashed/2
ms/2
no_style_check/0
no_style_check/1
nodebug/0
nospy/1
nospyall/0
notrace/0
spy/1
style_check/0
style_check/1
trace/0

fread/4

formatted read of a term

fread(*Format*, *FieldWidth*, *Modifier*, *Term*)

+Format	<atom> in the domain {a,b,f,i,n,r,s}.
+FieldWidth	<integer> in the range [-255..255]
+Modifier	<integer> in the range [-255..255]
-Term	<variable>

Read a simple term *Term* from the current input stream using the *Format*, *FieldWidth* and *Modifier* flag. The allowed formats are:

a	atom (uses modifier)
b	byte list (uses modifier)
f	floating point number (uses modifier)
i	integer
n	unsigned integer
r	arbitrary radix (uses modifier)
s	string (uses modifier)

A field width of zero gives free format input for byte lists, atoms and strings. This means input of a whole line up to and including a carriage return character. A negative modifier flag performs tab expansion and control character filtering in these cases.

See Also
fwrite/4

Formatted reading of atoms

fread(*Format*, *FieldWidth*, *Modifier*, *Term*)

fixed width

If the field width is a positive or negative integer, the number of characters read from the current input stream is the absolute value of the field width.

free width

When the field width is zero, a whole line up to and including a carriage return character is input.

read input literally

When the modifier flag is set to 0 or a positive integer and the field width is 0, the input is read in literally. Any carriage returns, line feeds, control characters and tabs present are included in the atom.

pre-process input

When the modifier is a negative integer and the field width is 0, the input is pre-processed. Carriage returns and line feeds are ignored, control characters are replaced by spaces and any tabs present are replaced with spaces up to the next eighth column.

Formatted reading of byte lists

fread(*Format*, *FieldWidth*, *Modifier*, *Term*)

fixed width

If the field width is a positive or negative integer, the number of characters read from the current input stream is the absolute value of the field width.

free width

When the field width is zero, a whole line up to and including a carriage return character is input.

read input literally

When the modifier flag is set to 0 or a positive integer and the field width is 0, the input is read in literally. Any carriage returns, line feeds, control characters and tabs present are included in the byte list.

pre-process input

When the modifier is a negative integer and the field width is 0, the input is pre-processed. Carriage returns and line feeds are ignored, control characters are replaced by spaces and any tabs present are replaced with spaces up to the next eighth column.

Formatted reading of floating point numbers

fread(*Format*, *FieldWidth*, *Modifier*, *Term*)

modifier

The *Modifier*, when applied to the f format, denotes the number of decimal places.

fixed field width

If the field width is a positive or negative integer, the number of characters read from the current input stream is the absolute value of the field width. Numbers must terminate with the specified number of decimal places within the given field width.

The modifier specifies the number of decimal places that should be found within the specified field width, if more decimal places are given *outside* the field width they are truncated from the returned number.

free field width

If the field width is zero, a single white-space delimited token is then read from the input stream and interpreted as a floating point number with the specified number of decimal places.

The modifier specifies the exact number of decimal places that the floating point number should have.

Formatted reading of integers

fread(*Format*, *FieldWidth*, *Modifier*, *Term*)

fixed width

If the field width is a positive or negative integer, the number of characters read from the current input stream is the absolute value of the field width. The number to be read in must be within the range 2147483648 to 2147483647.

free width

If the field width is zero, a single white-space delimited token is then read from the input stream and interpreted as an integer.

modifier

The modifier has no effect on the i format.

Formatted reading of unsigned integers

fread(*Format*, *FieldWidth*, *Modifier*, *Term*)

fixed width

If the field width is a positive or negative integer, the number of characters read from the current input stream is the absolute value of the field width. The number read in must be within the range 0 to 4294967295.

free width

If the field width is zero, a single white-space delimited token is then read from the input stream and interpreted as an unsigned integer.

modifier

The modifier has no effect on the n format.

Formatted reading of numbers in a given radix

fread(*Format*, *FieldWidth*, *Modifier*, *Term*)

modifier

The modifier, when applied to the r format, denotes the base of the number to be read in. The radix must be in the range 2 to 36.

The following number is read in as a hexadecimal number and converted to a decimal number.

fixed width

If the field width is a positive or negative integer, the number of characters read from the current input stream is the absolute value of the field width.

free width

If the field width is zero, a single white-space delimited token is then read from the input stream and interpreted as a number in the given radix.

Formatted reading of strings

fread(*Format*, *FieldWidth*, *Modifier*, *Term*)

fixed width

If the field width is a positive or negative integer, the number of characters read from the current input stream is the absolute value of the field width.

free width

When the field width is zero, a whole line up to and including a carriage return character is input.

read input literally

When the modifier flag is set to 0 or a positive integer and the field width is 0, the input is read in literally. Any carriage returns, line feeds, control characters and tabs present are included in the atom.

pre-process input

When the modifier is a negative integer and the field width is 0, the input is pre-processed. Carriage returns and line feeds are ignored, control characters are replaced by spaces and any tabs present are replaced with spaces up to the next eighth column.

free/9

return the free space available in Prolog's memory areas

free(*Backtrack*, *Local*, *Reset*, *Heap*, *Text*, *Program*, *System*, *Input*, *Output*)

-Backtrack	<variable>
-Local	<variable>
-Reset	<variable>
-Heap	<variable>
-Text	<variable>
-Program	<variable>
-System	<variable>
-Input	<variable>
-Output	<variable>

Return the number of bytes of free space available in *Backtrack*, *Local*, *Reset*, *Heap*, *Text*, *Program* and *System* spaces, and the *Input* and *Output* string buffers. The values returned are the amounts of memory free at the moment *free*/9 is called, without calling the garbage collector. There may well be more heap space potentially free, than is actually shown. If you want to know exactly how much heap space is available, you should make a call to the garbage collector, just prior to calling *free*/9.

See Also

garbage_collect/0
garbage_collect/1
gc/0
nogc/0
statistics/0
statistics/2
stats/4
total/9
ver/1
ver/4

functor/3

the relationship between a term its functor name and its arity

functor(*Term*, *Functor*, *Arity*)

?Term	<term> or <variable>
?Functor	<atom> or <variable>
?Arity	<integer> or <variable>

Succeeds if *Term* is a term with the specified *Functor* and *Arity*.

If *Term* is an uninstantiated variable, then *Functor* must be an atom and *Arity* must be an integer greater than or equal to 0. If *Arity* is 0, *Term* will be bound to the term *Functor*. If *Arity* is greater than 0, then *Term* will be bound to a compound term whose functor is the term *Functor*, and with arity *Arity*. Each argument of this compound term will be a distinct unbound variable.

See Also

=../2
arg/3
call/1
call/2
mem/3
one/1

fwrite/4

formatted write of a term

fwrite(*Format*, *FieldWidth*, *Modifier*, *Term*)

+Format	<atom> in the domain {a,b,f,i,n,r,s}.
+FieldWidth	<integer> in the range [-255..255]
+Modifier	<integer> in the range [-255..255]
+Term	<term>

Writes a simple term *Term* to the current output stream using the *Format*, *FieldWidth* and *Modifier* flag. The allowed formats are:

a	atom
b	byte list
f	floating point number (uses modifier)
i	integer
n	unsigned integer
r	arbitrary radix (uses modifier)
s	string

A field width of zero gives free format output for byte lists, atoms and strings. This means output is not limited to a specific field width. A negative modifier flag performs tab expansion and control character filtering.

See Also
fread/4

Formatted writing of atoms

fwrite(*Format*, *FieldWidth*, *Modifier*, *Term*)

fixed width

If the field width is a positive or negative integer, the absolute value of this integer indicates the number of characters to be written to the current output stream. If the length of the term to be written is less than this value, the output is 'padded out' with spaces up to the specified field width.

left justified

If the field width is a negative integer, the term is output left justified.

right justified

If the field width is a positive integer, the term is output right justified.

free width

When the field width is 0, there is no restriction of the output to a given field width, and there is no 'padding out' with spaces.

truncated output

When the modifier flag is set to 0 or a positive integer and the field width is fixed, if the term to be output is larger than this field width an error will be generated. Any carriage returns, line feeds, control characters and tabs present are also output.

However when the modifier is a negative integer and the field width is fixed, if the output is larger than the field, instead of generating an error, the output is truncated.

Formatted writing of byte lists

fwrite(*Format*, *FieldWidth*, *Modifier*, *Term*)

fixed width

If the field width is a positive or negative integer, the number of characters written to the current output stream is the absolute value of the integer. If the length of the term to be written is less than the absolute value of the field width, the output is 'padded out' with spaces up to the specified field width.

left justified

If the field width is a negative integer, the term is output left justified.

right justified

If the field width is a positive integer, the term is output right justified.

free width

When the field width is 0, there is no restriction of the output to a given field width, and there is no 'padding out' with spaces.

truncated output

When the modifier flag is set to 0 or a positive integer and the field width is fixed, if the term to be output is larger than this field width an error will be generated. Any carriage returns, line feeds, control characters and tabs present are also output.

However when the modifier is a negative integer and the field width is fixed, if the output is larger than the field, instead of generating an error, the output is truncated.

Formatted writing of floating point numbers

fwrite(*Format*, *FieldWidth*, *Modifier*, *Term*)

modifier

The modifier, when applied to the f format, denotes the number of decimal places.

fixed field width

The field width is a positive or negative integer. The numbers that are output contain the specified number of decimal places.

left justified

If the field width is a negative integer, the term is output left justified.

right justified

If the field width is a positive integer, the term is output right justified.

truncated output

When the field width is fixed and the modifier is a negative integer then the ouput is truncated to fit within the specified field width.

free field width

When the field width is 0, there is no restriction of the output to a given field width, and there is no 'padding out' with spaces.

Formatted writing of integers

fwrite(*Format*, *FieldWidth*, *Modifier*, *Term*)

fixed width

If the field width is a positive or negative integer, the number of characters output to the current output stream is the absolute value of the field width. The number to be output must be within the range 2147483648 to 2147483647.

left justified

If the field width is a negative integer, the term is output left justified.

right justified

If the field width is a positive integer, the term is output right justified.

free width

When the field width is 0, there is no restriction of the output to a given field width, and there is no 'padding out' with spaces.

truncated output

If the field width is fixed and the integer to be output is too large to fit within this field width, the integer may be truncated using a negative modifier. Otherwise an error will be generated.

Formatted writing of unsigned integers

fwrite(*Format*, *FieldWidth*, *Modifier*, *Term*)

fixed width

If the field width is a positive or negative integer. The number of characters written to the current output stream is the absolute value of the field width. The number ouput must be within the range 0 to 4294967295.

left justified

If the field width is a negative integer, the term is output left justified.

right justified

If the field width is a positive integer, the term is output right justified.

free width

When the field width is 0, there is no restriction of the output to a given field width, and there is no 'padding out' with spaces.

truncated output

If the field width is fixed and the integer to be output is too large to fit within this field width, the integer may be truncated using a negative modifier. Otherwise an error will be generated.

Formatted writing of numbers with a given radix

fwrite(*Format*, *FieldWidth*, *Modifier*, *Term*)

modifier

The modifier, when applied to the r format, denotes the base of the number to be output. The radix must be in the range 2 to 36.

The following number is converted from a decimal number and output as a hexadecimal number.

fixed width

If the field width is a positive or negative integer, the number of characters written to the current output stream is the absolute value of the field width. The number is output with leading zeroes up to the field width.

free width

When the field width is 0, there is no restriction of the output to a given field width, and there is no 'padding out' with spaces.

truncated output

If the field width is fixed and the number to be output is too large to fit within this field width, the number may be truncated using a negative modifier. Otherwise an error will be generated.

Formatted writing of strings

fwrite(*Format*, *FieldWidth*, *Modifier*, *Term*)

fixed width

If the field width is a positive or negative integer, the number of characters written to the current output stream is the absolute value of the field width. If the length of the term to be written is less than the absolute value of the field width, the output is 'padded out' with spaces up to the specified field width.

left justified

If the field width is a negative integer, the term is output left justified.

right justified

If the field width is a positive integer, the term is output right justified.

free width

When the field width is 0, there is no restriction of the output to a given field width, and there is no 'padding out' with spaces.

truncated output

When the modifier flag is set to 0 or a positive integer and the field width is fixed, if the term to be output is larger than this field width an error will be generated. Any carriage returns line feeds, control characters and tabs present are also output.

However when the modifier is a negative integer and the field width is fixed, if the output is larger than the field, instead of generating an error, the output is truncated.

garbage_collect/0

invokes the garbage collector explicitly

The predicate *garbage_collect*/0 invokes the immediate garbage collection of the memory area specified in the prolog flag gc_collect. This may be set to either heap or text. The garbage collection is done only if garbage collection is enabled (see *gc*/0 and *nogc*/0).

Note: when a program is running automatic garbage collection will always function regardless of the status of the *gc*/0 and *nogc*/0 flags.

See Also

free/9
garbage_collect/1
gc/0
nogc/0
statistics/0
statistics/2
stats/4
total/9
ver/1
ver/4

garbage_collect/1

invoke the garbage collection of the given memory area

garbage_collect(*Type*)

+Type	<atom>

If *Type* is the atom *heap* then the Prolog heap will be garbage collected. If *Type* is the atom *text* then the text space will be garbage collected. Note that garbage collection must be enabled (see *gc*/0 and *nogc*/0).

Note: when a program is running automatic garbage collection will always function regardless of the status of the *gc*/0 and *nogc*/0 flags.

See Also

free/9
garbage_collect/0
gc/0
nogc/0
statistics/0
statistics/2
stats/4
total/9
ver/1
ver/4

gc/0

enable the garbage collector

The predicate *gc*/0 enables explicit calls to the garbage collector.

Note: when a program is running automatic garbage collection will always function regardless of the status of the *gc*/0 and *nogc*/0 flags.

See Also
free/9
garbage_collect/0
garbage_collect/1
nogc/0
statistics/0
statistics/2
stats/4
total/9
ver/1
ver/4

get/1

read a non-white-space character from the current input stream

get(*N*)

?N	<variable> or <char>

Reads the next non-white space character from the current input stream, and unifies *N* with the ASCII value of this character. A white space character is defined to be one whose ASCII value is less than or equal to 32.

See Also

get0/1
getb/1
getx/2
put/1
putb/1
putx/2

get0/1

read a character from the current input stream

get0(*N*)

?N	<variable> or <char>

Reads a character from the current input stream, and unifies *N* with the ASCII value of this character. When the input file pointer is at the end of a file this *get0*/1 returns the value 1.

See Also

get/1
getb/1
getx/2
put/1
putb/1
putx/2

getb/1

get a byte direct from keyboard

getb(*Byte*)

-Byte	<variable>

Input a byte from the keyboard or mouse. IBM PC function keys are returned as negative integers, and mouse keys return -1, -2 and -3 for the pressing of the left, right and both buttons respectively.

When an IBM PC function key or cursor key is pressed, it generates a pair of bytes, the first of which is always the null byte (0), and the second an integer that refers to the actual key pressed. The predicates *getb*/1 and *grab*/1 return these pairs as the negated value of the second byte. For example: the IBM PC function key F1 generates the pair of numbers 0 and 59; the predicate *getb*/1 would return this value as -59.

When *getb*/1 is used Prolog waits for a key or mouse click.

See Also

get/1
get0/1
getx/2
put/1
putb/1
putx/2

getx/2

input a byte, word or dword

getx(*Size*, *Value*).

+*Size*	<integer> in the domain {0,1,2,4}
-*Value*	<variable>

Get *Value*, a binary integer with the given *Size*, from the current input stream. *Size* represents the number of bytes input. If *Size* is 1 then a single byte is read. If *Size* is 2 then a word (2 byte integer) is read. If *Size* is 4 then a dword (4 byte integer) is read. If *Size* is 0 then a single byte is read from the input stream without being consumed. *Value* is read in from the file in intel format (i.e, the least significant byte goes before the most significant byte)

This predicate is useful when getting input from files that come from other applications which use standard encoding of two and four byte integers.

See Also

get/1
get0/1
getb/1
put/1
putb/1
putx/2

grab/1

check for a byte direct from keyboard

grab(*Byte*)

-Byte <variable>

Input a byte from the keyboard or mouse. IBM function keys are returned as negative integers, and mouse keys return -1, -2 and -3 for the pressing of the left, right and both buttons respectively.

When an IBM PC function key or cursor key is pressed, it generates a pair of bytes, the first of which is always the null byte (0), and the second an integer that refers to the actual key pressed. The predicates *getb*/1 and *grab*/1 return these pairs as the negated value of the second byte. For example: the IBM PC function key F1 generates the pair of numbers 0 and 59; the predicate *grab*/1 would return this value as -59.

When *grab*/1 is used Prolog does *not* wait for a key or mouse click. If no key or mouse button is being pressed at the time of the call then *grab*/1 simply fails.

See Also

keys/1
ttyflush/0
ttyget/1
ttyget0/1
ttynl/0
ttyput/1
ttyskip/1
ttytab/1

ground/1

test for completely bound terms

ground(*Term*)

+Term	<term>

The predicate *ground*/1 succeeds if *Term* is currently instantiated to a term that is completely bound (has no uninstantiated variables in it); otherwise it fails.

See Also

atom/1
atomic/1
callable/1
char/1
chars/1
compound/1
float/1
integer/1
integer_bound/3
nonvar/1
number/1
simple/1
string/1
type/2
unifiable/2
var/1

halt/0

terminate the current Prolog session

Terminates the current Prolog session with an exit code of 0 and returns to the operating system (or the calling program).

The predicate *halt/0* is equivalent to the following program:

```
halt :-
    halt(0).
```

See Also

!/0
./2
->/2
:/2
\+/1
abort/0
break/0
break_hook/1
fail/0
false/0
halt/1
not/1
otherwise/0
repeat/0
repeat/1
true/0

halt/1

terminate the current Prolog session with a return code

halt(*Status*)

+Status	<integer> in the range [0..255]

Terminates the current Prolog session and returns control to the operating system (or the calling program). All files and windows are closed, and the screen is reset.

Status is the return code passed to the parent process. It can be tested from a DOS batch file using IF ERRORLEVEL command. *Status* must be an integer in the range 0 to 255 (inclusive). An exit code of 255 will also close down the current Windows session. This can be useful where an application is being built automatically from a DOS batch file.

If there are files open when *halt*/0 is called, any buffered file output will be written to disk and the files closed. If a file buffer cannot be output (e.g. the disk is full), an error will be generated and the attempt to halt will fail. The file is still closed however (without the updates), which means that a subsequent call to *halt*/0 will succeed.

During a halt the Prolog system shuts down all its currently open DLLs, fonts, menus, icons, bitmaps and Windows metafiles.

See Also

!/0
,/2
->/2
;/2
\+/1
abort/0
break/0
break_hook/1
fail/0
false/0
halt/0
not/1
otherwise/0
repeat/0
repeat/1
true/0

help/3

perform a windows help function

help(*HelpFile, Function, Context*)

+HelpFile	<atom>
+Function	<integer> in the range [1..7]
+Context	<integer>

Access the Windows help file using the specified function and context number. The *HelpFile* argument is an atom specifying the help file name and extension. The *Function* argument is a help function value. The *Context* argument is either an integer that is a context number for a particular topic in the help file or 0.

index/2

declare multiple argument indexes

index(*Pred*, *Indexes*)

+Pred	<pred_spec>
+Indexes	<list_of <integer> >

This predicate is only available in the Developer and Programmer editions of Prolog.

The predicate *index*/2 is a declaration recognised by the optimizing compiler to create multiple argument indexes in the optimized code. It specifies the arguments to be indexed (*Indexes*) in the given predicate (*Pred*). This declaration is used in conjunction with the predicates *optimize*/1 and *optimize_files*/1.

See Also
optimize/1
optimize_files/1

initialization/1

declare a goal to be run on loading a file

initialization(*Goal*)

+Goal <goal>

The predicate *initialization*/1 is defined as built-in prefix operator. It declares a *Goal* that is to be run when the file in which the declaration appears is loaded. It can be used as either a directive, or as part of a goal called at compile-time.

The initialization goal will be run after the loading of the file is completed and all other directives appearing in the file have been run.

See Also

abolish_files/1
compile/1
consult/1
ensure_loaded/1
load_files/1
load_files/2
multifile/1
prolog_load_context/2
reconsult/1
save_predicates/2
source_file/1
source_file/2
source_file/3

inpos/1

set the input stream position

inpos(*Position*)

?Position	<integer> or <variable>

The *inpos*/1 predicate allows you to reposition the file pointer associated with the current input stream. It can also be used to find the current position within the current input stream.

If *Position* is a non-negative integer, *inpos*/1 will move the file pointer associated with the current input file to be *Position* bytes from the beginning of the file.

If *Position* is a variable, it will be bound to an integer that is the current value of the file pointer associated with the current input file. This value represents the byte offset of the file pointer relative to the beginning of the file (offset 0).

Note: *inpos/1* can also reposition the read pointer in input strings.

See Also

at_end_of_file/0
at_end_of_line/0
find/1
flush/0
outpos/1
skip/1
skip_layout/0
skip_line/0
skip_term/0
stream_position/2
stream_position/3

input/1

set input from a file, device or string

input(*Stream*)

?Stream	<atom> or <integer> in the domain {0,1,2} or (<string>,<integer>) or <variable>

The predicate *input*/1 allows you to directly set or get the current input *Stream*. If the *Stream* argument is an atom it should name a currently open file (to check the names of currently open files see *fdict*/1). If the *Stream* argument is an integer in the domain {0,1,2} it names one of the special input streams. If the *Stream* argument is a conjunction of a string and an offset, the string is used for direct input starting at the character following the specified offset. If the *Stream* argument is a variable it will be bound to the currently set input stream. If the current input stream is a string the *Stream* argument will be bound to a conjunction of the entire input string and the current offset within the string.

See Also

output/1
see/1
seeing/1
seen/0
tell/1
telling/1
told/0

integer/1

test for an integer

integer(*Term*)

?Term	<term>

Succeeds if *Term* is instantiated to an integer. It fails if *Term* is any other type of term.

See Also

atom/1
atomic/1
callable/1
char/1
chars/1
compound/1
float/1
ground/1
integer_bound/3
nonvar/1
number/1
simple/1
string/1
type/2
unifiable/2
var/1

integer_bound/3

generate or test a number between lower and upper bounds

integer_bound(*Lower*, *Number*, *Upper*)

+Lower	<integer>
?Number	<integer> or <variable>
+Upper	<integer>

The predicate *integer_bound/3* Succeeds if *Number* is an integer between *Lower* and *Upper* inclusively. If *Number* is unbound *integer_bound*/3 will generate successive values for *Number* that lie between *Lower* and *Upper*.

See Also

atom/1
atomic/1
callable/1
char/1
chars/1
compound/1
float/1
ground/1
integer/1
nonvar/1
number/1
simple/1
string/1
type/2
unifiable/2
var/1

is/2

expression evaluator

X is *Expression*

?X	<number>
+Expression	<expr>

Evaluates *Expression* and unifies the result with *X*. There are two types of function supported in LPA Prolog: the generic arithmetic functions that will work with both floating point and integer values (making conversions where necessary) and the bitwise operators which only work on integer values and will generate an error if any other type of input is given (including floating point numbers).

See Also

</2
=:=/2
=</2
=\=/2
>/2
>=/2
seed/1

key_hook/3

system defined handler for handling control keys

key_hook(*Win*, *Key, Goal*)

+Win	<window_handle>
+Key	<integer>
+Goal	<term>

Invoke the system defined behaviour for handling the given *Key* in the window *Win* and then run the given *Goal*.

See Also

break_hook/1
change_hook/3
dll_hook/3
error_hook/2

keys/1

return the status of the system keys

keys(*Keys*)

-Keys <variable>

Return the current status of the keyboard system keys. The *Keys* argument is a 16-bit key value representing the state of the system shift, ctrl and alt keys as well as the lock modes.

See Also
grab/1
ttyflush/0
ttyget/1
ttyget0/1
ttynl/0
ttyput/1
ttyskip/1
ttytab/1

keysort/2

sort a list of key-value pairs into ascending order

keysort(*List1*, *List2*)

+List1	<list>
-List2	<variable>

Sorts the list *List1* into ascending order according to the standard order of terms. The sorted list is bound to the variable *List2*.

Each element of *List1* must be a term of the form:

```
Key-Value
```

The list will be sorted on the value of *Key*.

See Also

=/2
==/2
@</2
@=</2
@>/2
@>=/2
\=/2
\==/2
cmp/3
compare/3
eqv/2
len/2
occurs_chk/2
sort/2
sort/3
subsumes_chk/2

lcall/4

call a dynamic link library function

lcall(*Library*, *Function*, *Input*, *Output*)

+Library	<atom>
+Function	<atom> or <string>
+Input	<atom> or <string>
-Output	<variable>

Calls the named dynamic link library with the given function and input text, returning the output text if the function succeeds. The input text may be a string or an atom; the type of the output string depends upon the type of the input string.

This call must only be made with dynamic link libraries prepared especially for LPA Prolog, since it makes specific assumptions about the number, type, and passing convention of argument parameters. The creation of suitable dynamic link libraries is discussed in more detail in the Prolog *for Windows Programming Guide*.

See Also

dll_hook/3
lclose/1
ldict/1
lopen/1
'?DLL?'/3
winapi/5

lclose/1

close a dynamic link library

lclose(*Library*)

+Library <atom>

Closes the named dynamic link library, freeing up the memory resources it occupies. Up to eight dynamic link libraries may be open at any one time, so it may be necessary to close one in order to open another.

See Also
dll_hook/3
lcall/4
ldict/1
lopen/1
'?DLL?'/3
winapi/5

ldict/1

return a list of all currently open dynamic link libraries

ldict(*Libraries*)

-Libraries <variable>

Returns the dictionary of currently open dynamic link libraries. When a dynamic link library is opened, its name is added automatically to the library dictionary. The name is removed when the library is closed.

See Also
dll_hook/3
lcall/4
lclose/1
lopen/1
'?DLL?'/3
winapi/5

leash/2

set the interaction with the debugger

leash(*Point*, *Ports*)

+Point	<atom> in the domain {head, body}
+Ports	<atom> or <list_of <atom> > where each <atom> in the domain {call,exit,redo,fail}

Sets the leashing mode for the given *Point* to the given *Ports*. The purpose of *leash*/2 is to allow you to speed up single-stepping (creeping) through a program by telling the debugger that it does not always need to wait for user input after printing a trace message.

Point is either of the atoms 'head' or 'body'. Ports is a list of the ports in the given *Point* to be leashed. Valid port names are: 'call', 'exit', 'redo' and 'fail'.

By default, all four ports are leashed in both the head and body. On arrival at a leashed port the debugger will stop to allow you to look at the execution state and decide what to do next. At unleashed ports the goal is displayed but program execution does not stop to allow user interaction.

See Also

debug/0
debug_hook/1
debugging/0
force/1
leashed/2
ms/2
no_style_check/0
no_style_check/1
nodebug/0
nospy/1
nospyall/0
notrace/0
spy/1
style_check/0
style_check/1
trace/0

leashed/2

test or get when interaction with the debugger will occur

leashed(*Point*, *Port*)

?Point	<atom> or <variable>
?Port	<atom> or <variable>

If *Point and Port* are given, *leashed*/2 says whether there is a leash on that *Port* for that *Point*. If either *Point* or *Port* are variables *leashed*/2 will backtrack through the various leashed ports.

See Also

debug/0
debug_hook/1
debugging/0
force/1
leash/2
ms/2
no_style_check/0
no_style_check/1
nodebug/0
nospy/1
nospyall/0
notrace/0
spy/1
style_check/0
style_check/1
trace/0

len/2

return length of a term

len(*Term*, *Length*)

?Term	<term> or <variable>
?Length	<integer> or <variable>

Get or check the *Length* of the given *Term* or if *Term* is a variable build a list of variables from the given *Length*. If *Term* is a list, *Length* will be unified with the number of items on the list. If *Term* is a tuple, *Length* will be unified with the number of arguments in the tuple plus 1 for the functor. If *Term* is an atom or string, *Length* will be unified with the number of characters in the atom or string. If *Term* is a number, *Length* will be bound to the number of characters contained in its printed form. If *Term* is a variable, it will be instantiated to a list of variables of the given *Length*.

See Also

=/2
==/2
@</2
@=</2
@>/2
@>=/2
\=/2
\==/2
cmp/3
compare/3
eqv/2
keysort/2
length/2
occurs_chk/2
sort/2
sort/3
subsumes_chk/2

length/2

get the length of a Prolog list

length(*Term*, *Length*)

?Term	<list>
?Length	<variable>

If *Term* is a list, *Length* will be unified with the number of items in the list. If *Length* is an integer and *Term* is unbound, *Term* will be unified with a list of variables of the specified *Length*.

See Also

append/3
len/2
mem/3
member/2
member/3
remove/3
removeall/3
reverse/2

library_directory/1

defines a library directory

library_directory(*DirSpec*)

?DirSpec <file_spec>

DirSpec is either an atom giving the path to a file, or *PathAlias*(*DirSpec*), where *PathAlias* is defined by a *file_search_path*/2 fact.

The dynamic, multifile *library_directory*/1 facts define directories to search when a file specification *library*(*File*) is expanded to the full path.

There are a set of predefined *library_directory*/1 facts, but users may also define their own libraries simply by asserting the appropriate *library_directory*/1. To locate a library file, the *library_directory*/1 facts are tried one by one in the same sequence they appear in the Prolog database.

The file_search_path mechanism is an extension of the library_directory scheme. See *file_search_path*/2.

See Also

absolute_file_name/2
absolute_file_name/3
attrib/2
cat/3
chdir/1
close/1
del/1
dir/3
drive/1
env/2
fclose/1
fcreate/3
fdict/1
file_search_path/2
fname/4
fopen/3
mkdir/1
open/2
ren/2
rmdir/1
stamp/1

listing/0

list all the dynamic clauses in the workspace to the current output stream

Lists all dynamic clauses in the workspace to the current output stream.

See Also

abolish/1
abolish/2
abolish_files/1
assert/1
asserta/1
assert/2
assertz/1
clause/2
clauses/2
clause/3
dynamic/1
dynamic_call/1
functor/3
listing/1
retract/1
retractall/1
retract/2
volatile/1

listing/1

list the specified dynamic predicates to the current output stream

listing(*Tolist*)

+Tolist <pred_specs>

Lists all of the dynamic clauses specified by *Tolist* to the current output stream. *listing/1* always succeeds. If *Tolist* is a predicate specification of the form *Predicate*/*N* (where *Predicate* is the name of a dynamic relation and *N* is an integer), then all clauses for *Predicate* with arity *N* are listed. If *Tolist* is a list of predicate names, then all of the specified predicates are listed.

See Also

abolish/1
abolish/2
abolish_files/1
assert/1
asserta/1
assert/2
assertz/1
clause/2
clauses/2
clause/3
dynamic/1
dynamic_call/1
functor/3
listing/0
retract/1
retractall/1
retract/2
volatile/1

load_files/1

load the specified Prolog source and/or object files.

load_files(*FileSpecs*)

+FileSpecs <file_specs>

The *load_files*/1 predicate reads Prolog clauses, in source or in compiled form, and adds them to the Prolog database, after first deleting any previous versions of the predicates they define. Clauses for a single predicate must all be in the same file unless that predicate is declared to be multifile.

FileSpecs can be a file specification of the form: *PathAlias*(*File*) where *PathAlias* is an alias that refers to a specified path and *File* is a file name relative to that path. *FileSpecs* can also be an atom of the form: *Path/File* where *Path* defines the path where the file *File* will be found. *FileSpecs* may also be a list containing combinations of the above.

Note: a ".PL" or ".PC" extension may be omitted in a file specification.

If the file contains directives, that is, terms with principal functor ':-'/1 or '?-'/1, then these are executed as they are encountered.

See Also

abolish_files/1
compile/1
consult/1
ensure_loaded/1
initialization/1
load_files/2
multifile/1
prolog_load_context/2
reconsult/1
save_predicates/2
source_file/1
source_file/2
source_file/3

load_files/2

load the specified Prolog source and/or object files using certain options.

load_files(*FileSpec*, *Options*)

+FileSpecs	<file_specs>
+Options	<list_of <compound_term> >

The predicate *load_files*/2 loads the clauses in a Prolog file into the Prolog database according to some specified options. The *FileSpecs* argument can be a file specification of the form: *PathAlias*(*File*) where *PathAlias* is an alias that refers to a specified path and *File* is a file name relative to that path. *FileSpecs* can also be an atom of the form: *Path/File* where *Path* defines the path where the file *File* will be found. *FileSpecs* may also be a list containing combinations of the above.

The *Options* argument is a list of load file options that affect the way the file is loaded.

If any Prolog source file being loaded contains directives, that is, terms with principal functor ':-'/1 or '?-'/1, then these are executed as they are encountered.

Directives that are to be saved in object code files must be declared initialisation directives using the predicate *initialization/1*. Initialisation directives are executed after the load.

See Also

abolish_files/1
compile/1
consult/1
ensure_loaded/1
initialization/1
load_files/1
multifile/1
prolog_load_context/2
reconsult/1
save_predicates/2
source_file/1
source_file/2
source_file/3

lopen/1

open a dynamic link library

lopen(*Library*)

+Library <atom>

Opens the named dynamic link library. Up to eight dynamic link libraries may be open at any one time, so it may be necessary to close one in order to open another.

See Also
dll_hook/3
lcall/4
lclose/1
ldict/1
'?DLL?'/3
winapi/5

lwrupr/2

convert between lower and upper case

lwrupr(Lower,Upper)

?*Lower*	<atom>, <string> or <variable>
?Upper	<atom>, <string> or <variable>

Take an atom or string and return its upper case equivalent, or take an atom or string and return its lower case equivalent. The type of the output depends upon the type of the input. Note that because mixed and fixed case character sets are not congruent, the conversion is not necessarily reversible. For this reason, the case where both arguments are given is not supported.

See Also

ansoem/2
lcall/4
lclose/1
ldict/1
'?DLL?'/3
winapi/5

mem/3

return the specified member of a term

mem(*Record*, *Path*, *Field*)

+Record	<compound_term>
+Path	<list_of <integer> >
-Field	<variable>

The *mem*/3 predicate is used to access arbitrary members of lists and compound terms. It binds *Field* to the sub-field of *Record* that is identified by *Path*.

The *Record* argument must be a compound term (i.e. a list or a tuple). Individual fields can themselves be lists or compound terms (sub-records), or they can be atomic data items. *Path* must be a list of integers which index into *Record*. The simplest path is the empty list [] which will return the entire record. A single entry path, of the form [*N*], where *N* is a positive integer, returns the *Nth* element of the record. A single entry path, of the form [*N*], where *N* is a negative integer, returns a list of all the elements following the *Nth* element of the record. If the record is a tuple, the first element of the record is the functor, and the 2nd element is the first argument of the compound term. If it is a list, the first element is the head of the list, the second is the head of the tail, and so on. *Field* must be an uninstantiated variable.

See Also

append/3
arg/3
functor/3
length/2
member/2
member/3
remove/3
removeall/3
reverse/2

member/2

get or check a member of a list

member(*Element*, *List*)

?Element	<term>
?List	<list> or <variable>

The term *Element* is a member of the list *List*.

If *Element* is instantiated, *member*/2 will check that it is a member of the list (if *List* is a variable, it will be bound to a list that contains *Element*).

If *Element* is a variable, it will be bound to the first element of *List*. On backtracking, it will be bound to successive elements of *List*.

See Also

append/3
length/2
mem/3
member/3
remove/3
removeall/3
reverse/2

member/3

get or check a member of a list and its position in the list

member(*Element*, *List*, *Position*)

?Element	<term>
?List	<list> or <variable>
?Position	<integer> or <variable>

If *Element* is instantiated a check is made to see if it is on *List* and its position on *List* is either checked or returned.

If *Element* and *Position* are both uninstantiated solutions for each are generated on backtracking.

Note: the numbering of elements on the list begins at 1.

See Also

append/3
length/2
mem/3
member/2
remove/3
removeall/3
reverse/2

message_box/3

create a message box and return a response

message_box(*Buttons*,*Message*,*Response*).

+Buttons	<atom> in the domain {ok, okcancel, yesno, yesnocancel}
+Message	<string>
+Response	<variable>

Display a *Message* in a window with the specified *Buttons* and return the selected button in *Response*. The returned selected button may be one of the following: ok, cancel, yes or no.

See Also
erase_status_box/0
status_box/1

'?MESSAGE?'/4

user-defined Prolog program which intercepts messages

'?MESSAGE?'(*Window*, *Message*, *Data*, *Goal*)

+Window	<window_handle>
+Message	<integer>
+Data	<integer>
+Goal	<goal>

The user defined message hook, if it is defined, is called whenever a message interrupts the system. The *Window* argument is the window that generated the message. The *Message* argument is the integer message value of the message that was generated. The *Data* argument is an integer value that is the data associated with the message and the *Goal* argument is the Prolog goal that was interrupted by the message.

To allow the default processing of messages you should call *message_hook/4*.

When defining a message hook that writes to the console, you should bear in mind that the act of writing to the console itself generates a message. Any message hook that writes to the console should filter out the message 5 (msg_change) for the console window (1,1) and pass these values straight on to the *message_hook/4* predicate.

See Also
message_hook/4

message_hook/4

system defined behaviour for handling messages

message_hook(*Win*, *Msg*, *Data*, *Goal*)

+Win	<window_handle>
+Msg	<integer>
+Data	<integer>
+Goal	<goal>

Invoke the system defined message hook for the given window, message, data and interrupted goal. The *Win* argument is the name of the window that generated the message. The *Msg* argument is the integer number of the message that was generated. The *Data* argument is an integer that gives any data associated with the message and the *Goal* argument is the Prolog goal that was interrupted by the message.

The predicate *message_hook/4* handles all the messages that occur normally in the Prolog environment. This predicate provided to allow programmatic access to the default system error handler in user-defined *'?MESSAGE?'/4* programs.

See Also
'?MESSAGE?'/4

mkdir/1

make a directory

mkdir(*Dir*)

+Dir <atom>

Makes a new directory called *Dir* (It is the same as the MKDIR command under DOS.). *Dir* must be an atom that names the directory. If *Dir* does not specify a drive name, the directory is created on the current drive. If *Dir* does not specify a path name, the directory is created in the current directory.

See Also

absolute_file_name/2
absolute_file_name/3
attrib/2
cat/3
chdir/1
close/1
del/1
dir/3
drive/1
env/2
fclose/1
fcreate/3
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
open/2
ren/2
rmdir/1
stamp/1

ms/2

time a given Prolog goal

ms(*Call*, *Time*)

+Call	<goal>
-Time	<variable>

Runs the given *Call* and returns the *Time* in milliseconds (10 ^ -3 seconds) it took the goal to run.

Note: the functionality of this predicate is somewhat hardware dependent, and as a result may not work consistently on all models of computer.

See Also

date/3
date/4
ticks/1
time/4

msgbox/4

display the message box

msgbox(*Title*, *Message*, *Style*, *Button*)

+Title	<string>
+Message	<atom> or <string>
+Style	<integer>
-Button	<variable>

Display a standard Windows message box with a given title, message and style returning the users response to the dialog. The *Title* argument is a string that sets the message boxs window caption. The *Message* argument is either an atom or string that is the message to be shown to the user. The *Style* argument is a message box style value that dictates which combination of predefined buttons, icons and modality is used in the message box. The *Button* argument is a variable that gets bound to an integer indicating which button was used to terminate the dialog.

The predicate succeeds whichever button is clicked, or when RETURN is pressed.

See Also

abtbox/3
change_hook/3
chgbox/3
dirbox/4
erase_status_box/0
find_hook/3
fndbox/2
message_box/3
status_box/1
sttbox/2

multifile/1

allow the specified predicates to be defined in more than one file

:- multifile *Predicates*

+Predicates <pred_specs>

The predicate *multifile*/1 is a built-in prefix operator. *Predicates* is a single predicate specification of the form *Name/Arity*, or a sequence of predicate specifications separated by commas. *Name* must be an atom and *Arity* an integer in the range 0 to 64.

By default, all clauses for a predicate are expected to come from just one file. This assists with reloading and debugging of code. Declaring a predicate multifile means that its clauses can be spread across several different files.

The multifile declaration should precede all the clauses for the specified predicate *P* in each file that contains clauses for *P*.

See Also

abolish_files/1
compile/1
consult/1
ensure_loaded/1
initialization/1
load_files/1
load_files/2
prolog_load_context/2
reconsult/1
save_predicates/2
source_file/1
source_file/2
source_file/3

name/2

convert between an atom or number and a byte list

name(*Atomic*, *List*)

?Atomic	<atom> or <number>
?List	<char_list> or <variable>

List is the list of ASCII character codes that represents the atomic term *Atomic*. *Atomic* must be a variable, number or atom, while *List* must be a variable or a list of character codes. One of either *Atomic* or *List* must be a non-variable.

If *Atomic* is an atom or number, *List* will be unified with the list of character codes that make up its print name.

If *Atomic* is an unbound variable, *List* must be a list of character codes. If the characters in *List* represent a number (either integer or floating point), *Atomic* will be unified with that number. Otherwise, *Atomic* will be bound to an atom that contains those characters exactly.

See Also

=../2
atom_chars/2
atom_string/2
copy_term/2
number_atom/2
number_chars/2
number_string/2
numbervars/3
string_chars/2

nl/0

start a new line on the current output stream

Start a new line on the current output stream. Writes a carriage return followed by line feed to the current stream.

See Also
tab/1

no_style_check/0

turn off all compile-time style checking

When all the style checking is turned off, files will be loaded without checking for: clauses containing a single instance of a named variable, procedures whose clauses are not all adjacent to one another in the file and multiple definitions of the same procedure in different files.

See Also

debug/0
debug_hook/1
debugging/0
force/1
leash/2
leashed/2
ms/2
no_style_check/1
nodebug/0
nospy/1
nospyall/0
notrace/0
spy/1
style_check/0
style_check/1
trace/0

no_style_check/1

turn off the specified style of compile-time style checking

no_style_check(*Type*)

+Type <atom>

Turn off the specified *Type* of style checking. The Type argument is a style checking value.

See Also

debug/0
debug_hook/1
debugging/0
force/1
leash/2
leashed/2
ms/2
no_style_check/0
nodebug/0
nospy/1
nospyall/0
notrace/0
spy/1
style_check/0
style_check/1
trace/0

nodebug/0

switch the debug mode to off

Switches the debug mode to be *off*. Spy points are not removed, but they will have no effect during an evaluation. (This predicate is a synonym for *notrace*/0)

To remove spy points use *nospy*/1 or *nospyall*/0.

See Also

debug/0
debug_hook/1
debugging/0
force/1
leash/2
leashed/2
ms/2
no_style_check/0
no_style_check/1
nospy/1
nospyall/0
notrace/0
spy/1
style_check/0
style_check/1
trace/0

nofileerrors/0

turn off the reporting of file error messages

Sets the 'fileerrors' flag to off in which error messages are not reported by *see*/1, *tell*/1 and *open*/2 if the specified file cannot be opened.

The 'fileerrors' flag is only enabled by an explicit call to *fileerrors*/0, or via *prolog_flag*/3 which can also be used to obtain the current value of the 'fileerrors' flag.

See Also

fileerrors/0
no_style_check/0
no_style_check/1
prolog_flag/2
prolog_flag/3
prompt/2
prompts/2
style_check/0
style_check/1
switch/2

nogc/0

disable the garbage collector

nogc/0 disables explicit calls to garbage collector.

Note: when a program is running automatic garbage collection will always function regardless of the status of the *gc*/0 and *nogc*/0 flags.

See Also
free/9
garbage_collect/0
garbage_collect/1
gc/0
statistics/0
statistics/2
stats/4
total/9
ver/1
ver/4

nonvar/1

test for a non-variable

nonvar(*Term*)

?Term <term>

Succeeds if, and only if, *Term* is instantiated. It is the opposite of *var*/1.

See Also

atom/1
atomic/1
callable/1
char/1
chars/1
compound/1
float/1
ground/1
integer/1
integer_bound/3
number/1
simple/1
string/1
type/2
unifiable/2
var/1

nospy/1

remove the spy points from the specified predicates

nospy(*Spied*)

+Spied <pred_specs>

Removes the spy points from all of the predicates specified by *Spied*.

If *Spied* is a list of predicate specifications *nospy*/1 will remove the spy points from all predicates named on that list. The predicate *nospy*/1 just succeeds if there are no clauses defined for *Spied*, or if the clauses do not have a spy point set. The predicate *nospy* is defined as a prefix operator, so you do not need to surround its argument with brackets.

See Also

debug/0
debug_hook/1
debugging/0
force/1
leash/2
leashed/2
ms/2
no_style_check/0
no_style_check/1
nodebug/0
nospyall/0
notrace/0
spy/1
style_check/0
style_check/1
trace/0

nospyall/0

remove all spy points

Removes all spy points currently set on predicates.

See Also

debug/0
debug_hook/1
debugging/0
force/1
leash/2
leashed/2
ms/2
no_style_check/0
no_style_check/1
nodebug/0
nospy/1
notrace/0
spy/1
style_check/0
style_check/1
trace/0

not/1

logical negation

not *Call*

+Call <goal>

Logical negation. This succeeds if *Call* fails. No variables in *Call* are bound as the result of its negation. It is declared as a prefix operator.

See Also

!/0
,/2
->/2
;/2
\+/1
abort/0
break/0
break_hook/1
fail/0
false/0
halt/0
halt/1
otherwise/0
repeat/0
repeat/1
true/0

notrace/0

turn the trace mode to off

Switches the trace mode to be *off*. Spy points are not removed, but they will have no effect during an evaluation (This predicate is a synonym for *nodebug/0*).

To remove spy points use *nospy*/1 or *nospyall*/0.

See Also

debug/0
debug_hook/1
debugging/0
force/1
leash/2
leashed/2
ms/2
no_style_check/0
no_style_check/1
nodebug/0
nospy/1
nospyall/0
spy/1
style_check/0
style_check/1
trace/0

number/1

test for a floating point number or integer

number(*Term*)

?Term	<term>

Succeeds if *Term* is currently instantiated to a number (i.e. an integer or float). Fails for any other type of term.

See Also

atom/1
atomic/1
callable/1
char/1
chars/1
compound/1
float/1
ground/1
integer/1
integer_bound/3
nonvar/1
simple/1
string/1
type/2
unifiable/2
var/1

number_atom/2

convert between a number and an atom

number_atom(*Number*, *Atom*)

?Number	<number> or <variable>
?Atom	<atom> or <variable>

Initially either *Atom* must be instantiated to an atom or *Number* must be instantiated to a number.

If *Number* is initially instantiated to a number, *Atom* will be unified with the atomic equivalent of its printed representation. If *Atom* is initially instantiated to an atom that corresponds to the correct syntax of a number, then *Number* will be bound to that number.

See Also

=../2
atom_chars/2
atom_string/2
copy_term/2
name/2
number_chars/2
number_string/2
numbervars/3
string_chars/2

number_chars/2

convert between numbers and a list of characters

number_chars(*Number*, *CharList*)

?Number	<number> or <variable>
?CharList	<char_list> or <variable>

Initially either *CharList* must be instantiated to a list of ASCII character codes (containing no variables) or *Number* must be instantiated to a number.

If *Number* is initially instantiated to a number, *CharList* will be unified with the list of ASCII character codes that form its printed representation. If *CharList* is initially instantiated to a list of ASCII character codes that correspond to the correct syntax of a number, then *Number* will be bound to that number.

See Also

=../2
atom_chars/2
atom_string/2
copy_term/2
name/2
number_atom/2
number_string/2
numbervars/3
string_chars/2

number_string/2

convert between a number and a string

number_chars(*Number*, *String*)

?Number	<number> or <variable>
?String	<string> or <variable>

Initially either *String* must be instantiated to a string or *Number* must be instantiated to a number.

If *Number* is initially instantiated to a number, *String* will be unified with the string equivalent of its printed representation. If *String* is initially instantiated to a string that corresponds to the correct syntax of a number, then *Number* will be bound to that number.

See Also

=../2
atom_chars/2
atom_string/2
copy_term/2
name/2
number_atom/2
number_chars/2
numbervars/3
string_chars/2

numbervars/3

instantiate the variables in a given term

numbervars(*Term*, *FirstVar*, *LastVar*)

+Term	<term>
+FirstVar	<integer>
?LastVar	<integer> or <variable>

numbervars/3 instantiates each of the variables in *Term* to a term of the form '$VAR'(N).

FirstVar must be an integer. That integer is used as the value of N for the first variable in *Term* (starting from the left). The second distinct variable in *Term* is given a value of N satisfying "N is *FirstVar*+1"; the third distinct variable gets the value *FirstVar*+2, and so on. The last variable in Term has the value *LastVar*-1.

See Also

=../2
atom_chars/2
atom_string/2
copy_term/2
name/2
number_atom/2
number_chars/2
number_string/2
string_chars/2

occurs_chk/2

occurs check

occurs_chk(*Term*, *Var*)

+Term	<term>
+Var	<variable>

The occurs check. Succeeds if *Var* occurs in the given *Term*.

See Also

=/2
==/2
@</2
@=</2
@>/2
@>=/2
\=/2
\==/2
cmp/3
compare/3
eqv/2
keysort/2
len/2
sort/2
sort/3
subsumes_chk/2

one/1

one solution meta-call.

one(*Call*)

+Call <goal>

Succeeds if *Call* is true, but has the side effect of restricting the evaluation of *Call* to just one solution. On backtracking, no further solutions to *Call* are sought.

Call may be bound to a conjunction or disjunction of goals.

The *one* predicate acts as a "snip" operation (i.e. a restricted form of cut). It cuts out alternative solutions to a single call, without cutting out backtracking in any other part of the current query. The *one* predicate provides useful control information when you know that there will be only one way of solving a call. It cuts out redundant search on backtracking.

See Also

=../2
arg/3
call/1
call/2
functor/3

op/3

declare an operator with a given precedence and type

op(*Precedence*, *Type*, *Name*)

+Precedence	<integer>
+Type	<atom>
+Name	<atom>

Declares an operator with a given type and precedence. The *Name* argument is an atom that is the name of the operator. The *Precedence* is an integer between 0 and 1199, and whose type is *Type*. *Name* may also be a list of atoms, in which case all of the atoms are declared to be operators of the specified precedence and type.

To find the operators currently in force, use *current_op*/3.

See Also
current_op/3

open/2

open a file with the given access mode

open(*FileSpec*, *Mode*).

+FileSpec	<file_spec>
+Mode	<atom> in the domain {read,write,append}

open/2 opens the file specified by *FileSpec* with the given access mode *Mode*. *FileSpec* can be a file specification of the form: *PathAlias*(*File*) where *PathAlias* is an alias that refers to a specified path and *File* is a file name relative to that path. *FileSpec* can also be an atom of the form: *Path/File* where *Path* defines the path where the file *File* will be found.

Mode should be one of the atoms: 'read', 'write' or 'append'. If *Mode* is 'read' the file specified by *FileSpec* is opened and the file pointer is set to the beginning of the file. If *Mode* is 'write' the file specified by *FileSpec* is created and the file pointer starts at the beginning of the file. Note: in 'write' mode any file already in existence that corresponds to the file specified by *FileSpec* will be overwritten by this call to *open*/2. If *Mode* is 'append', and the file specified by *FileSpec* exists, it is opened and the file pointer is set to the end of the file, if the file specified by *FileSpec* does not exist then 'append' mode acts like 'write' mode.

See Also

absolute_file_name/2
absolute_file_name/3
attrib/2
cat/3
chdir/1
close/1
del/1
dir/3
drive/1
env/2
fclose/1
fcreate/3
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
ren/2
rmdir/1
stamp/1

optimize/1

optimize a static predicate

optimize(*Pred*)

+Pred <pred_spec>

This predicate is only available in the Developer and Programmer editions of Prolog.

Optimize the clauses for a static predicate specified by *Pred*. This predicate can be used in conjunction with the declaration *index*/2 to generate multiple argument indexing for the specified relation.

See Also
index/2
optimize_files/1

optimize_files/1

file to file optimization of code

optimize_files(*Files*)

+Files	<file_spec> - <file_spec>	or a list of such pairs

This predicate is only available in the Developer and Programmer editions of Prolog.

Optimize Prolog source code files into object code files. *optimize_files*/1 invokes the built-in, file-to-file optimizing compiler. *Files* is a list of pairs of the form: *Source* - *Object*, where *Source* represents the input source file and *Object* represents the destination object file. This predicate can be used in conjunction with the declaration *index*/2 to generate multiple argument indexing for each specified relation.

See Also
index/2
optimize/1

otherwise/0

succeed

Same as true. Always succeeds. (It is useful for laying out nested conditionals.).

See Also

!/0
./2
->/2
:/2
\+/1
abort/0
break/0
break_hook/1
fail/0
false/0
halt/0
halt/1
not/1
repeat/0
repeat/1
true/0

outpos/1

sets the output stream position

outpos(*Position*)

?Position	<integer> or <variable>

The *outpos*/1 predicate allows you to reposition the file pointer associated with the current output file stream. It can also be used to find the current position within the current output file stream.

If *Position* is a non-negative integer, *outpos*/1 will move the file pointer associated with current output file to be *Position* bytes from the beginning of the file.

If *Position* is a variable, it will be bound to an integer that is the current value of the file pointer associated with current output file. This value represents the byte offset of the file pointer relative to the beginning of the file (offset 0).

See Also

at_end_of_file/0
at_end_of_line/0
find/1
flush/0
inpos/1
skip/1
skip_layout/0
skip_line/0
skip_term/0
stream_position/2
stream_position/3

output/1

set output to the screen, a file or a string

output(*Stream*)

?Stream	<atom> or <integer> in the domain {0,1,2,3} or (<string>,<integer>) or <variable>

The predicate *output*/1 allows you to directly set or get the current output stream *Stream.* If the *Stream* argument is an atom it should name a currently open file. This file should have been opened using *fopen/3* in either read/write or write only access. If the *Stream* argument is an integer in the domain {0,1,2,3} it names one of the special output streams. If the *Stream* argument is a conjunction of a string plus an offset, the string is used for direct output with the output position set to the character following the offset. If the *Stream* argument is a variable it will be bound to the currently set output stream. If the current output stream is a string the *Stream* argument will be bound to a conjunction of the entire string plus the current offset within that string.

See Also

input/1
see/1
seeing/1
seen/0
tell/1
telling/1
told/0

pdict/3

return a dictionary of predicates

pdict(*Type*, *Flags*, *Dictionary*)

+Type	<integer> in the domain [-1,0,1,2,3,4]
+Flags	<integer>
-Dictionary	<variable>

Return a dictionary of predicates that match the given type and have the specified flag pattern. The *Type* argument is a predicate type value. The *Flags* argument is a flag pattern that is used internally. A value of -1 returns all predicates of the specified type. The *Dictionary* argument is a variable that is bound to a list of predicates that match the given type. Each predicate on the list is represented as a comma pair of the form: (*Predicate*,*Arguments*), where *Predicate* and *Arguments* are respectively the functor and arity of the predicate.

See Also

current_atom/1
current_op/3
current_predicate/1
current_predicate/2
def/3
defs/2
predicate_property/2

phrase/2

check if a sequence of symbols can be parsed as a given type

phrase(*Phrase*, *List*)

+Phrase	<atom> or <compound_term>
?List	<list> or <variable>

The predicate *phrase*/2 invokes the grammar rules in order to parse a sequence of symbols. It succeeds if *List* is a phrase of type *Phrase* (according to the current grammar rules).

The *Phrase* argument must be the name of a non-terminal symbol, or a grammar rule body. *phrase/2* will succeed if *List* represents a phrase of type *Phrase*. If *List* is a variable, *phrase/2* will attempt to generate a phrase of type *Phrase* and bind it to *List*.

See Also

'C'/3
expand_term/2
phrase/3

phrase/3

check if a sequence of symbols can be parsed as a given type

phrase(*Phrase*, *List*, *Rest*)

+Phrase	<atom> or <compound_term>
?List	<list> or <variable>
?Rest	<list> or <variable>

The *phrase*/3 predicate succeeds when the list *List* starts with a phrase of type *Phrase*, according to the currently defined grammar rules. *Rest* is that part of *List* that is left over after you have found the phrase.

See Also
'C'/3
expand_term/2
phrase/2

portray_clause/1

write a clause to the current output stream in listing format

portray_clause(*Clause*)

+Clause <clause>

Writes the clause *Clause* to the current output stream in the same format as the *listing* predicate. The clause will be terminated with a full stop and white space character.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
print/1
printq/1
prompt/2
read/1
skip_term/0
sysops/0
vars/2
write/1
write_canonical/1
writeq/1

predicate_property/2

find the association between predicates and properties

predicate_property(*Predicate*, *Property*)

?Predicate	<atom> or <compound_term> or <variable>
?Property	<atom> or <variable>

Get or test the relationship between predicates and their properties. If the *Predicate* argument is a variable then it will be bound to a predicate that has the given property. If the *Property* argument is a variable it will be bound to a predicate property for the given predicate. If both the *Predicate* and *Property* arguments are bound then a check is made to see if the specified predicate has the specified property. If both arguments are variables the program will backtrack through the various alternatives for the current Prolog database.

See Also

current_atom/1
current_op/3
current_predicate/1
current_predicate/2
def/3
defs/2
pdict/3

print/1

print a term to the current output stream

print(*Term*)

?Term <term>

The predicate *print*/1 writes *Term* to the current output stream. By default, the effect of this predicate is the same as that of *write*/1, but you can change its effect by providing clauses for the user-defined predicate *portray*/1.

If *Term* is a variable, then it is printed using *write*(*Term*). Otherwise the user-definable procedure *portray*(*Term*) is called. If this succeeds, then it is assumed that *Term* has been printed and *print*/1 exits (succeeds).

If the call to *portray*/1 fails, and if *Term* is a compound term, then *write*/1 is used to write the principal functor of *Term* and *print*/1 is called recursively on its arguments. If *Term* is atomic, it is written using *write*/1.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
printq/1
prompt/2
read/1
skip_term/0
sysops/0
vars/2
write/1
write_canonical/1
writeq/1

printq/1

print a quoted term to the current output stream

printq(*Term*)

?Term	<term>

The predicate *printq*/1 writes the given term to the current output stream. By default, the effect of this predicate is the same as that of *writeq*/1, but you can change its effect by providing clauses for the predicate *portray*/1.

If *Term* is a variable, then it is printed using *writeq*(*Term*). Otherwise the user-definable procedure *portray*(*Term*) is called. If this succeeds, then it is assumed that *Term* has been printed and *printq*/1 exits (succeeds).

If the call to *portray*/1 fails, and if *Term* is a compound term, then *writeq*/1 is used to write the principal functor of *Term* and *printq*/1 is called recursively on its arguments. If *Term* is atomic, it is written using *writeq*/1.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
prompt/2
read/1
skip_term/0
sysops/0
vars/2
write/1
write_canonical/1
writeq/1

profile/4

get or set a profile string

profile(*IniFile*, *Section*, *Entry*, *String*)

+IniFile	<atom> or <string>
+Section	<atom> or <string>
+Entry	<atom> or <string>
?String	<variable>

Read or write a *String* into the named initialisation file *IniFile*, using the given *Section* and *Entry*. If the given file, section or label does not exist, it is created automatically. If the given *Entry* is the empty string, the *Section* is deleted; if the given *String* is empty, the *Entry* is deleted. Note: the *IniFile* is not opened as a Prolog file or stream, but is maintained entirely by Windows.

prolog_flag/2

get or check the values for global environment variables

prolog_flag(*Flagname*, *Value*)

?Flagname	<atom>
?Value	<atom> or <variable>

The *prolog_flag/2* predicate allows you to test or retrieve the values of the global environment flags.

See Also

fileerrors/0
no_style_check/0
no_style_check/1
nofileerrors/0
prolog_flag/3
prompt/2
prompts/2
style_check/0
style_check/1
switch/2

prolog_flag/3

set and get values for global environment variables

prolog_flag(*Flagname*, *Old_value*, *New_value*)

+Flagname	<atom>
?Old_value	<atom> or <variable>
+New_value	<atom>

The *prolog_flag/3* predicate allows you to set the values of the global environment flags.

See Also

fileerrors/0
no_style_check/0
no_style_check/1
nofileerrors/0
prolog_flag/2
prompt/2
prompts/2
style_check/0
style_check/1
switch/2

prolog_load_context/2

find the context of the current load

prolog_load_context(*Key*, *Value*)

?Key	<atom> in the domain {module,file,stream,directory} or <variable>
?Value	<atom> or <variable>

Get or check the context of the current load, where *Key* and its related *Value* is a particular attribute of the load context. You can call *prolog_load_context*/2 from an embedded command in a file to find out the context of the current load. If called outside the context of a load, it simply fails. If *Key* and *Value* are variables *prolog_load_context*/2 will backtrack to find alternate solutions.

See Also

abolish_files/1
compile/1
consult/1
ensure_loaded/1
initialization/1
load_files/1
load_files/2
multifile/1
reconsult/1
save_predicates/2
source_file/1
source_file/2
source_file/3

prompt/2

get or set the Prolog prompt

prompt(*Old*, *New*)

-Old	<variable>
+New	<atom>

This predicate may be used to query or change the sequence of characters that indicate that the system is waiting for user input. The atom representing the old prompt is unified with *Old*. *New* must be an atom, and specifies the new prompt to be used.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
prompts/2
read/1
skip_term/0
sysops/0
vars/2
write/1
write_canonical/1
writeq/1

prompts/2

get or set the buffered console input prompts

prompts(*Initial*, *Continuation*)

?Initial	<variable>	or <atom>
?Continuation	<variable>	or <atom>

Get the *Initial* and *Continuation* atoms representing the prompts used by the buffered console input routines, or set new prompts from the given atoms.

See Also

fileerrors/0
no_style_check/0
no_style_check/1
nofileerrors/0
prolog_flag/2
prolog_flag/3
prompt/2
style_check/0
style_check/1
switch/2

put/1

write an ASCII character to the current output stream

put(*N*)

+N	<char>

Writes the character whose ASCII code is *N* to the current output stream. *N* can be an integer in the ASCII range (0 to 255), or an expression that evaluates to an integer in the ASCII range.

If *N* does not evaluate to an integer in the ASCII range, *put* will generate an error.

If the current output stream is a window, and *N* is the ASCII code of a control character, the character will not be output literally. Instead the control character will be interpreted by the window management system.

These control characters only affect the screen if the current output stream is a window, or it is the standard output stream. If the current stream is a file, the characters are output literally.

See Also

get/1
get0/1
getb/1
getx/2
putb/1
putx/2

putb/1

byte output direct to screen

putb(*Byte*)

+Byte <char>

Output to the screen the ASCII character related to the ASCII value *Byte* . If *Byte* is a negative integer then two characters are output to the screen: the first is the null character (0), followed by the character related to the absolute value of *Byte*. This facility is provided primarily for symmetry with the predicate *getb*/1.

See Also

get/1
get0/1
getb/1
getx/2
put/1
putx/2

putx/2

output a byte, word or dword to the current output stream

putx(*Size*, *Value*)

+*Size*	<integer> in the domain {1,2,4}
+*Value*	<integer>

Output *Value*, a binary integer, to the current output stream with the given *Size*. *Size* represents the number of bytes output. If *Size* is 1 then *Value* can be in the range [0..255], which corresponds to a byte. If *Size* is 2 then *Value* can be in the range [0..65535], which corresponds to a word (2 byte integer). If *Size* is 4 then *Value* can be in the range [2147483648..2147483647], which corresponds to a dword (4 byte integer). The binary integer represented by *Value* is output in intel format (i.e, the least significant byte goes before the most significant byte)

This predicate is useful when outputting to files from other applications which use standard encoding of two and four byte integers. For example, a dBASE III header contains a pointer to the end of file which is represented as a dword. To output a dword in the correct format you would use the following call to *putx*/2:

```
?- putx(4, Val).
```

where *Val* should be instantiated to an integer.

See Also

get/1
get0/1
getb/1
getx/2
put/1
putb/1

read/1

read a term from the current input stream

read(*Term*)

?Term	<variable> or <atom>

Reads the next term from the current input stream and unifies with *Term*. In the input stream, the term must be followed by a dot ('**.**') and at least one white space character (i.e. a character whose ASCII code is less than or equal to 32). The dot and white space character are read in but are not considered part of the term.

If end of file is encountered at the beginning of the read, *Term* will be unified with the atom *end_of_file*.

Note: user input is prompted by the currently defined prompt (see *prompt/2*), the default prompt is a vertical bar character followed by a colon.

If the *Term* read by *read/1* is not a valid term normally an error is generated. This behaviour can be modified, however, using the Prolog flag syntax_errors.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
prompt/2
skip_term/0
sysops/0
vars/2
write/1
write_canonical/1
writeq/1

reconsult/1

load a source code program into memory replacing the previous version

reconsult(*FileSpec*)

+FileSpec <file_specs>

A synonym for *consult*/1.

Loads the source program in the file *FileSpec*. *FileSpec* can be a list of files to consult.

FileSpec can be a file specification of the form: *PathAlias*(*File*) where *PathAlias* is an alias that refers to a specified path and *File* is a file name relative to that path. *FileSpec* can also be an atom of the form: *Path/File* where *Path* defines the path where the file *File* will be found.

If *FileSpec* does not specify a file extension then '.PL' is assumed.

Any commands in *FileSpec* are executed as they are encountered. A command is a term with functor *':-'*.

If *FileSpec* is the atom *user*, clauses and commands are entered from the terminal using the built-in editor.

See Also

abolish_files/1
compile/1
consult/1
ensure_loaded/1
initialization/1
load_files/1
load_files/2
multifile/1
prolog_load_context/2
save_predicates/2
source_file/1
source_file/2
source_file/3

remove/3

remove an element from a list

remove(*Element*, *List*, *Remainder*)

?Element	<term>
?List	<list> or <variable>
?Remainder	<list> or <variable>

If *Element* and *List* are instantiated then the *Element* is removed from the *List* to give *Remainder*.

If *Remainder* and *List* are instantiated then an attempt is made to instantiate *Element* to a single element of *List* which when removed gives the list *Remainder*. If this cannot be done *remove*/3 fails.

See Also

append/3
length/2
mem/3
member/2
member/3
removeall/3
reverse/2

removeall/3

remove all occurrences of an item from a list

removeall(*Item*, *List*, *Remainder*)

?Item	<term> or <variable>
+List	<list>
?Remainder	<list> or <variable>

Remove all occurrences of *Item* from the *List* to leave *Remainder*.

See Also

append/3
length/2
mem/3
member/2
member/3
remove/3
reverse/2

ren/2

rename a file

ren(*Old*, *New*)

+Old	<atom>
+New	<atom>

Renames the disk file *Old* to have the name *New*. Both *Old* and *New* must be atoms. *Old* must be the name of an existing file. *New* must be the name of a file that does not yet exist. If *Old* and *New* specify a disk drive then they must both be the same. *ren*/2 can be used to move a file from one directory to another (provided both directories are on the same disk).

See Also

absolute_file_name/2
absolute_file_name/3
attrib/2
cat/3
chdir/1
close/1
del/1
dir/3
drive/1
env/2
fclose/1
fcreate/3
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
rmdir/1
stamp/1

repeat/0

succeed even on backtracking.

Succeeds when called and on backtracking. Any calls which textually precede the *repeat* in the body of a clause will never be reached on backtracking.

repeat is defined as:

```
repeat.
repeat :- repeat.
```

See Also

!/0
./2
->/2
:/2
\+/1
abort/0
break/0
break_hook/1
fail/0
false/0
halt/0
halt/1
not/1
otherwise/0
repeat/1
true/0

repeat/1

succeed even on backtracking for a given number of times

repeat(*Number*)

+Number <integer>

Succeeds when initially called, and succeeds for the given *Number* of times on backtracking. Any calls which textually precede the *repeat* in the body of a clause will not be reached until the *repeat*/1 predicate has been backtracked into the given *Number* of times.

See Also

!/0
./2
->/2
:/2
\+/1
abort/0
break/0
break_hook/1
fail/0
false/0
halt/0
halt/1
not/1
otherwise/0
repeat/0
true/0

retract/1

delete a clause that matches the given clause

retract(*Clause*)

+Clause <clause>

Searches for the first clause in the database that matches *Clause*. If such a clause is found, it is deleted. Any variables in *Clause* are bound as a result of the unification. This predicate is non-deterministic. On backtracking there is an attempt to find and delete another matching clause. This search always starts at the beginning of the list of clauses for the relation name of *Clause*. So all clauses asserted between the retract and the redo of the call (even if added using *asserta*/1) are candidates for deletion on the redo. The call to *retract*/1 fails when there are no (more) clauses that match *Clause*.

Please note: Modifying dynamic code while it is running can lead to unpredictable behaviour.

See Also

abolish/1
abolish/2
abolish_files/1
assert/1
asserta/1
assert/2
assertz/1
clause/2
clauses/2
clause/3
dynamic/1
dynamic_call/1
functor/3
listing/0
listing/1
retractall/1
retract/2
volatile/1

retract/2

retract a clause at a specified position

retract(*Pred*, *Pos*)

+Pred	<pred_spec>	
+Pos	<integer>	> 0

Retract clause at *Pos* for the dynamic predicate specified by *Pred*. If *Pred* is not a dynamic predicate, or if Pos is greater than the number of clauses defining *Pred retract*/2 fails.

Please note: Modifying dynamic code while it is running can lead to unpredictable behaviour.

See Also

abolish/1
abolish/2
abolish_files/1
assert/1
asserta/1
assert/2
assertz/1
clause/2
clauses/2
clause/3
dynamic/1
dynamic_call/1
functor/3
listing/0
listing/1
retract/1
retractall/1
volatile/1

retractall/1

delete all clauses that match the given clause head

retractall(*Head*)

+Head <compound_term>

Deletes every clause in the database whose head matches *Head*. Variables in *Head* are left uninstantiated by the call. On backtracking there is no attempt to redo the call, even though matching clauses may have been asserted.

Head must represent a call to a dynamic predicate.

Please note: Modifying dynamic code while it is running can lead to unpredictable behaviour.

See Also

abolish/1
abolish/2
abolish_files/1
assert/1
asserta/1
assert/2
assertz/1
clause/2
clauses/2
clause/3
dynamic/1
dynamic_call/1
functor/3
listing/0
listing/1
retract/1
retract/2
volatile/1

reverse/2

check or get the reverse of a list

reverse(*List*, *Revlist*)

?List	<list> or <variable>
?Revlist	<list> or <variable>

If *List* and *Revlist* are instantiated, a check is made to see if one is a reverse of the other.

If either of *List* or *Revlist* is a variable, it is bound to a reverse of the other.

See Also

append/3
length/2
mem/3
member/2
member/3
remove/3
removeall/3

rmdir/1

delete a directory

rmdir(*Dir*)

+Dir <atom>

Deletes the directory called *Dir* (provided it is empty). (It is the same as the RMDIR command under DOS.)

Dir must be an atom that names the directory. If *Dir* does not specify a drive name, the directory is assumed to be on the current drive. If *Dir* does not specify a path name, it is assumed to be in the current directory.

You cannot delete the root directory (\), the current directory, nor any non-empty directory.

See Also

absolute_file_name/2
absolute_file_name/3
attrib/2
cat/3
chdir/1
close/1
del/1
dir/3
drive/1
env/2
fclose/1
fcreate/3
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
ren/2
stamp/1

save_predicates/2

save the specified predicates to a file in object code format.

save_predicates(*ListOfPredSpecs*, *FileSpec*)

+ListOfPredSpecs	<pred_specs>
+FileSpec	<file_spec>

This predicate is only available in the Developer and Programmer editions of Prolog.

The *save_predicates*/2 program saves the current definitions of all the predicates specified by *ListOfPredSpecs* into a file in object code format.

ListOfPredSpecs is a list of the predicates to be saved. The predicate specifications are of the form *predicate/arity*. *FileSpec* is the name of the target file in which the object code will be saved.

FileSpec can be a file specification of the form: *PathAlias*(*File*) where *PathAlias* is an alias that refers to a specified path and *File* is a file name relative to that path. *FileSpec* can also be an atom of the form: *Path/File* where *Path* defines the path where the file *File* will be found. The extension '.PC' is assumed if none is given.

See Also

abolish_files/1
compile/1
consult/1
ensure_loaded/1
initialization/1
load_files/1
load_files/2
multifile/1
prolog_load_context/2
reconsult/1
source_file/1
source_file/2
source_file/3

see/1

set the current input stream

see(*Stream*)

+Stream <file_spec>

Causes *Stream* to become the current input stream. If *Stream* is the name of an open input stream, it is made the current input stream. Otherwise the specified file is opened (for read access only) and made the current input stream. If it is not possible to open a file called *Stream*, an error will be generated. *see*/1 does not attempt to create a new file.

To make the user's terminal the current input stream, *Stream* should be the reserved stream name *user*. The *user* stream always uses the built-in editor for user input, and it echoes this input in the current window.

Streams are not automatically closed. You should close them with *seen*/0 or *close*/1.

See Also

input/1
output/1
seeing/1
seen/0
tell/1
telling/1
told/0

seed/1

re-seed the internal random number generator

seed(*Number*)

+Number <number>

Resets the seed for the random number generator to be *Seed*.

This predicate is useful when developing simulations that require the same sequence of random numbers to be generated on different occasions.

See Also

</2
=:=/2
=</2
=\=/2
>/2
>=/2
is/2

seeing/1

return the current input stream

seeing(*Stream*)

-*Stream* <variable>

Stream is unified with the name of the current input stream.

See Also
input/1
output/1
see/1
seen/0
tell/1
telling/1
told/0

seen/0

reset the current input stream to the standard input stream

The current input stream is reset to *user* immediately. If the original input stream was a disk file, then it is closed.

See Also
input/1
output/1
see/1
seeing/1
tell/1
telling/1
told/0

setof/3

find the set of instances of a term for which a Prolog goal is true

setof(*Term*, *Call*, *List*)

?Term	<term>
+Call	<goal>
?List	<variable>

Succeeds if *List* is a non-empty list of instances of *Term* such that *Call* is provable. *Term* may be any type of Prolog term, *Call* must be a goal of the form:

V1^V2^ ...^Vn^Goal (n ≥ 0)

where *V1*,*V2*,...,*Vn* are variables in the call to solve *Goal*. They are the existentially quantified variables of *Goal*.

The solution set, *List*, will be sorted according to the standard ordering used by the *compare*/3 predicate. *List* will not contain any duplicate entries.

If *Call* contains any free variables, then *setof*/3 will generate alternative sets for distinct instantiations of the free variables. A free variable is one which appears in *Call* but does not appear in *Term* and is not existentially quantified. For example in the call:

```
setof(A, B^C^ f(A,B,C,D,E), S)
```

D and E are free variables.

If there are no free variables, then *setof*/3 will have at most one solution. Otherwise a separate set will be generated for different instantiations of the free variables. These alternative sets are returned on backtracking until no further sets can be found.

See Also

^/2
bagof/3
findall/3
forall/2
solution/2

show_dialog/1

See Also
call_dialog/2
window_handler/2
window_handler/4
wdcreate/7

simple/1

test for an atom, number or variable

simple(*Term*)

+Term <term>

simple/1 succeeds if *Term* is currently instantiated to either an atom, a number or a variable; otherwise it fails.

See Also

atom/1
atomic/1
callable/1
char/1
chars/1
compound/1
float/1
ground/1
integer/1
integer_bound/3
nonvar/1
number/1
string/1
type/2
unifiable/2
var/1

skip/1

skip to just after the specified ASCII value on the current input stream

skip(*N*)

+N	<integer_expr> in the range [0..255]

Reads (from the current input stream) all characters up to and including the character whose ASCII value is *N*.

N can be an integer in the ASCII range (0 - 255). It can also be an arithmetic expression that evaluates to an integer in the ASCII range. This means that it can be a string of the form "*c*" which evaluates to the ASCII code of the character *c*.

See Also

at_end_of_file/0
at_end_of_line/0
find/1
flush/0
inpos/1
outpos/1
skip_layout/0
skip_line/0
skip_term/0
stream_position/2
stream_position/3

skip_layout/0

skip past the white space characters on the current input stream

Skip past all of the ASCII layout (white space) characters (in the range [0..32]) on the current input stream.

See Also
at_end_of_file/0
at_end_of_line/0
find/1
flush/0
inpos/1
outpos/1
skip/1
skip_line/0
skip_term/0
stream_position/2
stream_position/3

skip_line/0

skip the remaining input characters of the current line

Comments Skip the remaining input characters of the current line on the current input stream. Coding with *skip_line*/0 and *at_end_of_line*/0 to handle line input is more portable among different operating systems than checking end of line by the input character code.

See Also

at_end_of_file/0
at_end_of_line/0
find/1
flush/0
inpos/1
outpos/1
skip/1
skip_layout/0
skip_term/0
stream_position/2
stream_position/3

skip_term/0

skip the remaining input characters up to the end of a term

Comments Skip to the end of a term on the current input stream.

See Also

at_end_of_file/0
at_end_of_line/0
find/1
flush/0
inpos/1
outpos/1
skip/1
skip_layout/0
skip_line/0
stream_position/2
stream_position/3

solution/2

return the nth soution to a specified call

solution(*Call*, *Nth*)

+Call	<goal>
?Nth	<integer> or <variable>

If *Nth* is an integer then the *Call* is executed and re-satisfied until the *Nth* solution has been found. If the number of solutions to the *Call* is less than the value of *Nth* then *solution*/2 will fail. If *Nth* is a variable then the *Call* is executed and, upon its initial success, *Nth* is instantiated to 1. Upon backtracking, *Nth* is successively instantiated to the next solution number.

See Also

^/2
bagof/3
findall/3
forall/2
setof/3

sort/2

sort a list into ascending order and remove duplicates

sort(*List1*, *List2*)

+List1	<list>
-List2	<variable>

Each member of the list *List1* is compared and sorted into ascending order according to the standard ordering of terms, and duplicate elements are removed. The sorted copy of *List1* is bound to the variable *List2*.

See Also

=/2
==/2
@</2
@=</2
@>/2
@>=/2
\=/2
\==/2
cmp/3
compare/3
eqv/2
keysort/2
len/2
occurs_chk/2
sort/3
subsumes_chk/2

sort/3

sort a list into ascending order using a key, do not remove duplicates

sort(*List1*, *List2*, *Path*)

+List1	<list>
-List2	<variable>
+Path	<list_of <integer> >

Sorts the list *List1* into ascending order using the sort key described by *Path*. The sorted copy of *List1* is bound to *List2*. This predicate is useful for sorting lists of records. It allows the records to be sorted on a particular field.

List1 should be a list of records to sort. A record is either a compound term or a list. *Path* must be a list of integers. It is used to identify a specific field of a record (i.e. a complete list or compound term). The empty list [] identifies a complete record. A single element list, of the form [*Element*], where *Element* is a positive integer identifies a particular field within a record. A single element list, of the form [*Element*], where *Element* is a negative integer, identifies the elements following that particular field of the record. For example, the list [1] refers to the functor of a compound term, or the first element of a list. The list [2] refers to the first argument of a compound term, or the second element of a list, and so on. A two element list identifies a field within a field. For example, given the record:

```
record([fred,bloggs], [birth, friday, 30, october])
```

The path [2,1] refers to the atom *fred*. The path [3,4] refers to *october*. Note the way in which the functor of a compound term is considered the first field of a record.

The *sort*/3 predicate will order the list of records in *List1* on the field identified by *Path*. The records will be sorted into ascending order according to the standard ordering of terms.

Note: duplicate elements will *not* be removed.

See Also

=/2
==/2
@</2
@=</2
@>/2
@>=/2
\=/2
\==/2
cmp/3
compare/3
eqv/2
keysort/2
len/2
occurs_chk/2
sort/2
subsumes_chk/2

source_file/1

check or get the files that are currently loaded

source_file(*FileSpec*)

?FileSpec	<file_spec> or <variable>

The predicate *source_file*/1 is true if *FileSpec* refers to a currently loaded file.

FileSpec can be a file specification of the form: *PathAlias*(*File*) where *PathAlias* is an alias that refers to a specified path and *File* is a file name relative to that path. *FileSpec* can also be an atom of the form: *Path/File* where *Path* defines the path where the file *File* will be found.

If *FileSpec* is unbound, it is successively unified with the absolute file names of all currently loaded files.

If *FileSpec* is not the name of a loaded file or there are no source files currently open then *source_file*/1 simply fails.

See Also

abolish_files/1
compile/1
consult/1
ensure_loaded/1
initialization/1
load_files/1
load_files/2
multifile/1
prolog_load_context/2
reconsult/1
save_predicates/2
source_file/2
source_file/3

source_file/2

check or get the predicates associated with currently loaded files

source_file(*PredSpec*, *FileSpec*)

?PredSpec	<pred_spec> or <variable>
?FileSpec	<file_spec> or <variable>

The predicate *source_file*/2 is true if *PredSpec* refers to a currently loaded predicate and *FileSpec* refers to the source file with which the predicate is associated. *PredSpec* may be the general term related to a predicate or a variable.

FileSpec can be a file specification of the form: *PathAlias*(*File*) where *PathAlias* is an alias that refers to a specified path and *File* is a file name relative to that path. *FileSpec* can also be an atom of the form: *Path/File* where *Path* defines the path where the file *File* will be found.

If *PredSpec* is instantiated to a general term related to a loaded predicate and *FileSpec* is a variable, then *FileSpec* will be successively unified with the absolute file names of the files in which the *PredSpec* is defined. If *PredSpec* is a variable and *FileSpec* refers to a loaded file then *PredSpec* will be successively unified with each of the predicates defined in that file.

See Also

abolish_files/1
compile/1
consult/1
ensure_loaded/1
initialization/1
load_files/1
load_files/2
multifile/1
prolog_load_context/2
reconsult/1
save_predicates/2
source_file/1
source_file/3

source_file/3

the same as source_file/2 but also returns the numbers of the clauses

source_file(*PredSpec*, *ClauseNumber*, *FileSpec*)

?PredSpec	<pred_specs> or <variable>
?ClauseNumber	<integer> or <variable>
?FileSpec	<file_spec> or <variable>

The predicate *source_file*/3 is true if clause number *ClauseNumber* of predicate *PredSpec* comes from the currently loaded file *FileSpec*. *PredSpec* may be a general term related to a predicate or a variable.

FileSpec can be a file specification of the form: *PathAlias*(*File*) where *PathAlias* is an alias that refers to a specified path and *File* is a file name relative to that path. *FileSpec* can also be an atom of the form: *Path/File* where *Path* defines the path where the file *File* will be found. Any combination of bound and unbound arguments is possible, and *source_file*/3 will generate the others.

See Also

abolish_files/1
compile/1
consult/1
ensure_loaded/1
initialization/1
load_files/1
load_files/2
multifile/1
prolog_load_context/2
reconsult/1
save_predicates/2
source_file/1
source_file/2

spy/1

set a spy point on the specified predicates

spy(*Tospy*)

+Tospy <pred_specs>

Sets a spy point on all of the predicates specified by *Tospy*.

Tospy must be either a single predicate specification of the form *Predicate*/*Arity* or a list of predicate specifications.

If there are no clauses defined for *Tospy*, *spy*/1 just succeeds. It does not report an error.

spy is a built-in prefix operator, so you do not need to surround its argument with brackets.

See Also

debug/0
debug_hook/1
debugging/0
force/1
leash/2
leashed/2
ms/2
no_style_check/0
no_style_check/1
nodebug/0
nospy/1
nospyall/0
notrace/0
style_check/0
style_check/1
trace/0

stamp/2

set or get a file date and time stamp

stamp(*File*, *Stamp*)

+File	<file_name>
?Stamp	<variable> or <integer>

Sets or gets the time and date.of the given *File* according to the *Stamp* argument. The Stamp argument is a 32-bit integer time value.

See Also

absolute_file_name/2
absolute_file_name/3
attrib/2
cat/3
chdir/1
close/1
del/1
dir/3
drive/1
env/2
fclose/1
fcreate/3
fdict/1
file_search_path/2
fname/4
fopen/3
library_directory/1
mkdir/1
open/2
ren/2
rmdir/1

statistics/0

display statistics about the current status of the system

Displays statistics relating to memory usage, showing the *Backtrack*, *Local*, *Reset*, *Heap*, *Text*, *Program* and *System* spaces, and the *Input* and *Output* string buffers. Also shown are statistics relating to time, the amount of elapsed time, the amount of time spent during active processing, the amount of time spent waiting for keyboard input and the amount of time spent garbage collecting. Note that *statistics/0* performs a garbage collection prior to calculating the information.

See Also

free/9
garbage_collect/0
garbage_collect/1
gc/0
nogc/0
statistics/2
stats/4
total/9
ver/1
ver/4

statistics/2

get individual memory statistics

statistics(*Keyword*, *Statistic*)

+Keyword	<atom>
-Statistic	<variable>

The *statistics*/2 predicate is used to obtain individual statistics on Prologs memory areas. The *Keyword* argument is a memory area keyword. The *Statistic* argument is a variable which becomes bound to the amount of free space available in the specified memory area. Note that *statistics/2* performs a garbage collection prior to calculating the information.

See Also

free/9
garbage_collect/0
garbage_collect/1
gc/0
nogc/0
statistics/0
stats/4
total/9
ver/1
ver/4

stats/4

return assorted runtime statistics

stats(*Elapsed*, *Idle*, *GCTime*, *GCCount*)

-Elapsed	<variable>
-Idle	<variable>
-GCTime	<variable>
-GCCount	<variable>

Return the total elapsed time, idle time, garbage collection time and garbage collection count for the current state of the Prolog system. The *Elapsed* argument is a variable that will be bound to the number of ticks (See *ticks/1*) since the start of the current Prolog session. The *Idle* argument is a variable that will be bound to the number of ticks spent waiting for user input. The *GCTime* argument is a variable that will be bound to the number of ticks spent garbage collecting the system. The *GCCount* argument is a variable that will be bound to the number of times the garbage collector has been called during the current session. Note that *stats/4* does not explicitly call the garbage collector.

See Also

free/9
garbage_collect/0
garbage_collect/1
gc/0
nogc/0
statistics/0
statistics/2
total/9
ver/1
ver/4

status_box/1

display a status message window

status_box(*Message*)

+Message <atom> or <string>

Display a status message in a window. This predicate is reccomended for its portability as it is available on both DOS and Windows platforms it is equivalent to calling *sttbox/2* with the second argument set to 0.

One useful feature of the status box is that only the input lines containing one or more characters replace the characters already in the status box. This means that, using multiple line input, you can refresh part of the status box without having to re-draw the whole window.

See Also

erase_status_box/0
message_box/3
sttbox/2

stream_position/2

get the current position of the specified stream

stream_position(*Stream*, *Pos*)

+Stream	<file_name>
-Pos	<variable>

stream_position/2 is true when *Stream* is an atom representing an open stream, *Pos* is bound to an integer that indicates the current position of the *Stream* pointer.

See Also

at_end_of_file/0
at_end_of_line/0
find/1
flush/0
inpos/1
outpos/1
skip/1
skip_layout/0
skip_line/0
skip_term/0
stream_position/3

stream_position/3

get the current position of the specified stream

stream_position(*Stream*, *Old*, *New*)

+Stream	<file_name>
-Old	<variable>
+New	<integer>

stream_position/3 unifies the current position of the read/write pointer for *Stream* with *Old*, then sets the position to *New*. *Old* and *New* are both integers.

Note that this operation only makes sense on streams that are connected to disk files. If Stream is any other type of stream, or is not a valid stream, *stream_position*/3 will generate an error.

See Also

at_end_of_file/0
at_end_of_line/0
find/1
flush/0
inpos/1
outpos/1
skip/1
skip_layout/0
skip_line/0
skip_term/0
stream_position/2

string/1

test for a string

string(*Term*)

?Term	<term>

Succeeds if *Term* is a Prolog string. Note: the Prolog string is a compact text data type which differs from the atom and byte-list types. See programming guide for a full description of the string data type.

See Also

atom/1
atomic/1
callable/1
char/1
chars/1
compound/1
float/1
ground/1
integer/1
integer_bound/3
nonvar/1
number/1
simple/1
type/2
unifiable/2
var/1

string_chars/2

convert between strings and character lists

string_chars(*String*, *CharList*)

?String	<string> or <variable>
?CharList	<char_list> or <variable>

Initially either *CharList* must be instantiated to a list of ASCII character codes (containing no variables) or *String* must be instantiated to a string.

If *String* is initially instantiated to a string, *CharList* will be unified with the list of ASCII character codes that represent its printed representation. If *CharList* is initially instantiated to a list of ASCII character codes then *String* will be unified with a string containing those characters.

See Also

=../2
atom_chars/2
atom_string/2
copy_term/2
name/2
number_atom/2
number_chars/2
number_string/2
numbervars/3

sttbox/2

display or update a status box

sttbox(*Text*, *FontAction*)

+Text	<atom> or <string>
+FontAction	<atom> or <integer> in the domain {-1,0,1,65535}

Display a standard modeless status box or update its contents with the given text. The *Text* argument may be a string or atom. The *FontAction* argument is either the name of a font (including the special numeric font names) to be used in the dialog, -1 in which case the status box goes away or 65535 in which case the Prolog bitmap is displayed as a background to the status box.

The status box has a special window handle of 2, which can be used in conjunction with the predicate *wenable/2* to enable or disable the status box window.

One useful feature of the status box is that only the input lines containing one or more characters replace the characters already in the status box. This means that, using multiple line input, you can refresh part of the status box without having to re-draw the whole window.

See Also

abtbox/3
change_hook/3
chgbox/3
dirbox/4
erase_status_box/0
find_hook/3
fndbox/2
message_box/3
msgbox/4
status_box/1

stuff/3

compress the data in the current input stream to the current output stream

stuff(*BufSize*, *RawCount*, *CompCount*).

+BufSize	<integer> in the range [0..4]
-RawCount	<variable>
-CompCount	<variable>

This predicate reads data from the current input stream, compressing them and outputting the compressed equivalent to the current output stream. It terminates when end of file is encountered on input. The *BufSize* argument is a buffer size value that specifies the size of the sliding window and the lookahead buffer. The *RawCount* argument is a variable that is bound to the total number of raw (uncompressed) bytes processed. The *CompCount* argument is a variable that is bound to the total number of compressed bytes processed.

In the LZSS algorithm, used by this compression predicate, what is known as a sliding window is maintained over recently output data, while a lookahead buffer peeks into the stream of data yet to be compressed. The contents of the look-ahead are compared with all locations in the sliding window, and if a match of two or more characters is found, the address and length of the match within the window, rather than the characters themselves, is output.

By experimenting with the sliding window size and comparing the last two values, it should be easy to determine the optimimum setting for whichever type of data you want to compress.

See Also
fluff/3

style_check/0

turn on all compile-time style checking.

The predicate *style_check*/0 turns on all the compile-time style checking of source files. This is equivalent to the call:

```
?- style_check(all).
```

A complete style check involves: checking for clauses containing a single instance of a named variable, checking for procedures whose clauses are not all adjacent to one another in the file and checking for multiple definitions of the same procedure in different files. See *no_style_check*/1.

See Also

debug/0
debug_hook/1
debugging/0
force/1
leash/2
leashed/2
ms/2
no_style_check/0
no_style_check/1
nodebug/0
nospy/1
nospyall/0
notrace/0
spy/1
style_check/1
trace/0

style_check/1

turn on the specified type of compile-time style checking.

style_check(*Type*)

+Type <atom>

Turn on the specified type of style checking. The *Type* argument is a style check value.

Since all style checking is on by default, this predicate is only used to put back style checking after it has been turned off by *no_style_check*/1.

See Also

debug/0
debug_hook/1
debugging/0
force/1
leash/2
leashed/2
ms/2
no_style_check/0
no_style_check/1
nodebug/0
nospy/1
nospyall/0
notrace/0
spy/1
style_check/0
trace/0

subsumes_chk/2

check that one term subsumes another

subsumes_chk(*General*, *Specific*)

+General <term>

+Specific <term>

Succeeds if *Specific* is a specific instance of the term *General*.

See Also

=/2
==/2
@</2
@=</2
@>/2
@>=/2
\=/2
\==/2
cmp/3
compare/3
eqv/2
keysort/2
len/2
occurs_chk/2
sort/2
sort/3

switch/2

set or get the value of Prolog command line switch

switch(*Switch*, *Value*)

+Switch	<atom>
?Value	<term> or <variable>

Set or get the *Value* of the *Switch*. Switches are denoted as a single char atom in the range 'a' to 'z'. Note: once Prolog is running switch values have no effect on the system itself. All the switches are therefore available to the user as a convenient store for up to 26 named integer values.

See Also

dos/0
dos/1
env/2
exec/3
ver/4

sysops/0

re-install all of the system-declared operators

The predicate *sysops*/0 re-installs all of the system-declared operators. Because operators can be re-defined it is useful to be able to re-install the default set of operators, so that standard Edinburgh syntax files may be loaded. Any user-defined operators will not be affected by a call to *sysops*/0.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
prompt/2
read/1
skip_term/0
vars/2
write/1
write_canonical/1
writeq/1

system_menu/3

invoke a development environment menu function

system_menu(*Window*, *Menu*, *Function*)

+Window	<window_handle>
+Menu	<atom>
+Function	<atom> or <compound_term>

Comments

The predicate *system_menu/3* runs the function from the menu using the specified window as the current focus. The *Window* argument should be the name of a window created by Prolog. The *Menu* argument should be an atom that is the name of one of the development environment menus. The *Function* argument is either an atom or a tuple that represents a valid environment function. All functions allow you to enter in as if the menu item had been just selected from the menu. Certain functions allow you to "answer" the dialog that would normally have been displayed by the menu item.

tab/1

write the given number of spaces to the current output stream

tab(*N*)

+N	<integer_expr> in the range [0..255]

Writes *N* spaces (ASCII code 32) to the current output stream. *N* may be an integer expression.

See Also
nl/0
write/1

tell/1

set the current output stream

tell(*Stream*)

+Stream <file_spec>

Causes *Stream* to become the current output stream. *Stream* must be an atom. If *Stream* is the name of an open stream (i.e. a reserved stream, or a window, or an open file), then it is made the current output stream. Otherwise a file called *Stream* is created (using *create*/1) and made the current output stream.

Streams are not automatically closed. You should close them with *told*/0 or *close*/1.

See Also

input/1
output/1
see/1
seeing/1
seen/0
telling/1
told/0

telling/1

return the current output stream

telling(*Stream*)

+Stream <variable>

Stream is unified with the name of the current output stream. Note: the output stream is set up with the *tell*/1 predicate and closed with the *told*/1 predicate.

See Also
input/1
output/1
see/1
seeing/1
seen/0
tell/1
told/0

term_expansion/2

user-defined hook for grammar rule translation

term_expansion(*Input*, *Output*)

+Input	<term>
+Output	<variable>

This user-defined predicate is used to change the way that Prolog translates grammar rules. The predicate *expand_term/2* calls *term_expansion/2* if it exists. If the call to *term_expansion/2* succeeds no further expansion is tried. If the call to *term_expansion/2* fails the normal grammar rule translation is then tried.

If after your own term expansion you want to additionally perform the default grammar rule expansion, you should use the predicate *expand_dcg/2*. This predicate converts grammar rules to Prolog without re-invoking *term_expansion/2*.

throw/2

throw the given error for the given goal

throw(*Error*, *Goal*)

+Error	<integer>
+Goal	<goal>

The predicate *throw/2* generates an actual *Error* for the specified *Goal*. If there is no call to *catch/2* remaining on the call stack, the *Goal* and the *Error* number is passed to the user-defined *'?ERROR?'/2,* if it exists, or otherwise to the default sytem *error_hook/2*.

See Also

abort/0
catch/2
catch/3
error_hook/2
error_message/2
flush/0
unknown_predicate_handler/2

ticks/1

get a time reference

ticks(*Time*)

-*Time* <variable>

Return the *Time* of the system ticker and hardware timer as a 32 bit integer to give a 1193181/256 Hz (approximately 1/4660 seconds) time reference. The count re-cycles roughly once per 10.5 days.

Because all 32 bits of the integer word are used in the value returned by *ticks*/1 this value will be negative during the second half of the 10.5 day cycle. The absolute positive values can be computed as the relative value modulo 2^32.

Note: the functionality of this predicate is somewhat hardware dependant, and as a result may not work properly on all models of computer.

See Also

date/3
date/4
ms/2
time/4

time/4

get the system time

time(*Hours*, *Minutes*, *Seconds*, *Hundredths*)

-Hours	<variable>
-Minutes	<variable>
-Seconds	<variable>
-Hundredths	<variable>

Return the system time in *Hours*, *Minutes*, *Seconds* and *Hundredths*. Note: the PC clock only has a resolution of around 55 ms, so the *Hundredths* variable will return values that are rounded to 1/18.2 seconds.

See Also

date/3
date/4
ms/2
ticks/1

timer/2

get or set the status of the given timer interrupt

timer(*Timer*, *Status*)

+Timer	<integer> in the range [0..63]
?Status	<variable>, <integer> or (<integer>, <integer>)

Get or set the status of the given timer. Timers are Prolog interrupts that can be set to go off after a given interval. There are 64 available timers which can be used by creating a definition for the *'?TIMER?'/3* hook. The *Timer* argument is an integer between 0 and 63 that specifies the timer. If the *Status* argument is bound it should be a timer status value. If the *Status* argument is a variable it will be bound to a timer status return value.

To set the status of a timer you must specify an interval in system ticks (see *ticks/1* for a more detailed description of the ticks unit) and an optional absolute base time. Setting an interval of 0 turns the timer off. The absolute base time for a timer should be specified as the number of ticks since the current Prolog session started. If you do not specify a base time the current elapsed time since the start of the session is used. A timer will go off when the absolute base time plus the interval has been reached or surpassed.

If you are setting timer goals to go off at regular intervals you can set the same interval for the next timer goal and set its absolute base time to be the absolute end time of the previous timer goal.

The interesting thing to note here is that *timer/2* returns the absolute time at which the given timer goal should have gone off and not the time it actually went off. So if, for some reason, a given timer goal gets delayed the next timer goal will go off as near to its intended time as it can. This means that in a sequence of regular timer goals any delay will be isolated to the particular timer goals that were delayed and will not be reflected onto the sequence as a whole.

See Also

timer_hook/3
'?TIMER?'/3
wait/1
wbusy/1
wflag/1

'?TIMER?'/3

user-defined hook for the 64 built-in timers

'?TIMER?'(*Timer*,*Interval*,*Goal*)

+Timer	<integer> in the range [0..63]
+Interval	<integer>
+Goal	<goal>

The *'?TIMER?'/3* program is used to handle an interrupt generated by one of the 64 built-in timers. The *Timer* argument will be matched with the timer that generated an interrupt. The *Interval* argument will be matched with the timer's (*Interval*,*BaseTime*) pair. A timer will go off when the absolute base time plus the interval has been reached or surpassed. The *Goal* argument is the Prolog goal that was interrupted by the timer. This hook should be used in conjunction with the predicate *timer/2*. To call the default behaviour for handling timers you should call the predicate *timer_hook/3*.

See Also

timer/2
timer_hook/3
wait/1
wbusy/1
wflag/1

timer_hook/3

built-in timer hook

break_hook(*Timer,Interval,Goal*)

+Timer	<integer> in the range [0..63]
+Interval	<integer>
+Goal	<goal>

Invoke the system defined timer hook with the given *Timer*, *Interval* and *Goal*. The default behaviour is to not reset the timer and simply run the *Goal*. This predicate is mainly provided to allow programmatic access to the default system timer handler in user-defined *'?TIMER?'/3* programs.

See Also

timer/2
'?TIMER?'/3
wait/1
wbusy/1
wflag/1

told/0

reset the current output stream to the standard output stream

This predicate resets the current output stream to the standard output stream (*user*). If the original input stream was a disk file, then the file is closed.

See Also

input/1
output/1
see/1
seeing/1
seen/0
tell/1
telling/1

total/9

return the total space allocated to Prolog's memory areas

total(*Backtrack*, *Local*, *Reset*, *Heap*, *Text*, *Program*, *System*, *Input*, *Output*)

-Backtrack	<variable>
-Local	<variable>
-Reset	<variable>
-Heap	<variable>
-Text	<variable>
-Program	<variable>
-System	<variable>
-Input	<variable>
-Output	<variable>

Return the number of bytes of total space allocated to the *Backtrack*, *Local*, *Reset*, *Heap*, *Text*, *Program* and *System* spaces, and the *Input* and *Output* string buffers. See the predicates *free/9*, *statistics/0* and *statistics/2* to find out how much of this memory is actually used.

See Also

free/9
garbage_collect/0
garbage_collect/1
gc/0
nogc/0
statistics/0
statistics/2
stats/4
ver/1
ver/4

trace/0

switch the trace mode to on

Switches the trace mode to *on*. In this mode the debugger stops at every dynamic predicate that is called, allowing you to single-step through the execution of an interpreted Prolog program.

The debugger will not be invoked if the top level query is preceded with *':-'*. Nor will it be invoked when executing queries in files that are being consulted.

See Also

debug/0
debug_hook/1
debugging/0
force/1
leash/2
leashed/2
ms/2
no_style_check/0
no_style_check/1
nodebug/0
nospy/1
nospyall/0
notrace/0
spy/1
style_check/0
style_check/1

true/0

succeed

The predicate *true*/0 is a useful "no operation" in a program it always succeeds.

See Also

!/0
./2
->/2
:/2
\+/1
abort/0
break/0
break_hook/1
fail/0
false/0
halt/0
halt/1
not/1
otherwise/0
repeat/0
repeat/1

ttyflush/0

flush the user output stream

Flush the current output buffer to the output stream. This predicate is particularly useful to force the immediate display of the characters in the output buffer, where otherwise output to the screen will only be performed when the output buffer is full.

See Also

grab/1
keys/1
ttyget/1
ttyget0/1
ttynl/0
ttyput/1
ttyskip/1
ttytab/1

ttyget/1

read a non-white-space character from the user input stream

ttyget(*N*)

?N	<variable> or <char>

Reads the next non-white space character from the user input stream, and unifies *N* with the ASCII value of this character. A white space character is defined to be one whose ASCII value is less than or equal to 32.

See Also

grab/1
keys/1
ttyflush/0
ttyget0/1
ttynl/0
ttyput/1
ttyskip/1
ttytab/1

ttyget0/1

read a character from the user input stream

ttyget0(*N*)

?N <variable> or <char>

Reads a character from the user input stream, and unifies *N* with the ASCII value of this character. When the input file pointer is at the end of a file this *ttyget0*/1 returns the value 1.

See Also
grab/1
keys/1
ttyflush/0
ttyget/1
ttynl/0
ttyput/1
ttyskip/1
ttytab/1

ttynl/0

start a new line on the user output stream

Start a new line on the user output stream. This predicate works by writing a carriage return followed by line feed to the user output stream.

See Also
grab/1
keys/1
ttyflush/0
ttyget/1
ttyget0/1
ttyput/1
ttyskip/1
ttytab/1

ttyput/1

write an ASCII character to the user output stream

ttyput(*N*)

+N	<char>

Writes the character whose ASCII code is *N* to the user output stream. *N* can be an integer in the ASCII range (0 to 255), or an expression that evaluates to an integer in the ASCII range.

If *N* does not evaluate to an integer in the ASCII range, *ttyput/1* will generate an error.

If *N* is the ASCII code of a control character, the character will not be output literally. Instead the control character will be interpreted by the window management system.

See Also

grab/1
keys/1
ttyflush/0
ttyget/1
ttyget0/1
ttynl/0
ttyskip/1
ttytab/1

ttyskip/1

skip to just after the specified ASCII value on the user input stream

ttyskip(*N*)

+N	<integer_expr> in the range [0..255]

Reads (from the user input stream) all characters up to and including the character whose ASCII value is *N*.

N can be an integer in the ASCII range (0 - 255). It can also be an arithmetic expression that evaluates to an integer in the ASCII range. This means that it can be a string of the form "*c*" which evaluates to the ASCII code of the character *c*.

See Also

grab/1
keys/1
ttyflush/0
ttyget/1
ttyget0/1
ttynl/0
ttyput/1
ttytab/1

ttytab/1

write the given number of spaces to the user output stream

ttytab(*N*)

+N	<integer_expr> in the range [0..255]

Writes *N* spaces (ASCII code 32) to the user output stream. *N* may be an integer expression.

See Also

grab/1
keys/1
ttyflush/0
ttyget/1
ttyget0/1
ttynl/0
ttyput/1
ttyskip/1
ttytab/1

type/2

return type of a term

type(*Term*, *Type*)

+Term	<term>
-Type	<variable>

Return the type of a given term. The *Type* argument is a term type value. The *Term* argument is any Prolog term.

The predicate *type*/2 can be used to increase the efficiency of programs that need to perform different tasks depending on the types of its input arguments. Instead of having a program that tests for each type in a separate clause, a program can be written that uses *type*/2 to determine the type of an argument, and then makes use of first argument indexing to switch to the appropriate code.

See Also

atom/1
atomic/1
callable/1
char/1
chars/1
compound/1
float/1
ground/1
integer/1
integer_bound/3
nonvar/1
number/1
simple/1
string/1
unifiable/2
var/1

unifiable/2

check that two terms are potentially unifiable

unifiable(*Term1*, *Term2*)

+Term1	<term>
+Term2	<term>

This predicate may be used to check that two terms are potentially unifiable without binding any of their arguments in the process.

See Also

atom/1
atomic/1
callable/1
char/1
chars/1
compound/1
float/1
ground/1
integer/1
integer_bound/3
nonvar/1
number/1
simple/1
string/1
type/2
var/1

unknown_predicate_handler/2

user defined fact that specifies the handling of unknown predicates

unknown_predicate_handler(*Unknown*, *Action*)

?Unknown	<goal> or <variable>
+Action	<goal>

If the *unknown_predicate_handler*/2 predicate is defined, when the goal specified in *Unknown* is called and there is no definition for that goal then the *Action* goal will be called.

If *Unknown* is a variable then every time a goal is called that has no definition then the *Action* goal will be called.

See Also

abort/0
catch/2
catch/3
error_hook/2
error_message/2
flush/0
throw/2

var/1

test for an uninstantiated variable

var(*Term*)

?Term	<term>

Succeeds if, and only if, *Term* is an uninstantiated variable. An uninstantiated variable is one which has not been bound to another term (other than another uninstantiated variable).

See Also

atom/1
atomic/1
callable/1
char/1
chars/1
compound/1
float/1
ground/1
integer/1
integer_bound/3
nonvar/1
number/1
simple/1
string/1
type/2
unifiable/2

vars/2

return a named list of vars in a term

vars(*Term*, *Vars*)

+Term	<term>
-Vars	<variable>

Returns *Vars*, a list of (*Name*, *Variable*) pairs for each af the variables occurring in the given *Term*. For any variable occuring only once in the Term, Name will be the single underscore character, '_'. Variables occuring more than once in the Term will be assigned names of the form 'A', 'B', 'C', etc. This predicate can be used in conjunction with the predicates *ewrite/2*, *ewrite/3*, *eprint*/2 and *eprint*/3 to output meaningful variable names, rather than the internal representation of the variables.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
prompt/2
read/1
skip_term/0
sysops/0
write/1
write_canonical/1
writeq/1

ver/1

output the standard banner

ver(*Mode*)

+Mode <integer>

Print the version banner in the given *Mode* to the current output stream. A full heap and text space garbage collection is performed, and the banner is displayed with the amounts of space remaining in the nine internal data areas. *Mode* may be 0 (no tramlines) or 1 (tramlines printed).

See Also

free/9
garbage_collect/0
garbage_collect/1
gc/0
nogc/0
statistics/0
statistics/2
stats/4
total/9
ver/4

ver/4

return information on the current version of Prolog

ver(*System*, *Version*, *Date*, *Serial_number*)

-System	<variable>
-Version	<variable>
-Date	<variable>
-Serial_number	<variable>

Return the system name, the current version, the date of creation and the serial number of your Prolog system.

See Also

free/9
garbage_collect/0
garbage_collect/1
gc/0
nogc/0
statistics/0
statistics/2
stats/4
total/9
ver/1

volatile/1

declare that the clauses for a predicate will not be saved in object files

:- volatile *PredSpec*

+PredSpec <pred_specs>

The predicate *volatile*/1 is a built-in prefix operator. When a predicate is defined as being volatile its clauses will not be saved in object code format files by Prolog 'save' predicates.

You can use *volatile*/1 at compile-time and at run-time. In both cases the predicate specified will be declared as volatile. When used as a compile-time directive, the volatile declaration of a predicate must appear before all clauses of that predicate.

You can check to see if a predicate is volatile by using *predicate_property*/2. The properties, as well as the predicate, can be deleted with *abolish*/1.

See Also

abolish/1
abolish/2
abolish_files/1
assert/1
asserta/1
assert/2
assertz/1
clause/2
clauses/2
clause/3
dynamic/1
dynamic_call/1
functor/3
listing/0
listing/1
retract/1
retractall/1
retract/2

wait/1

get or set the window message status

wait(*Flag*)

+Flag	<variable> or <integer>	in the domain {0,1}

Yield to Windows for the given type of wait time. The *Flag* argument can be a wait type value or a variable. If *Flag* is a variable this predicate returns the number of messages pending in the message queue.

Normally LPA Prolog will yield control to Windows for one message cycle every 256th predicate call. Screen and keyboard I/O also yields to Windows for the duration of the I/O, as does the *beep/2* predicate. In optimized code where, due to the optimization, fewer predicate calls are made LPA Prolog yields to Windows less often. In the case of repeat - fail loops, because neither *repeat/0* or *fail/0* make any predicate calls, the onus of yielding to Windows is on the code inside the loop. If this code is also optimized there may be no yielding to windows while the loop is in operation, effectively 'locking out' the Windows environment until the loop has finished. To modify this behaviour *wait/1* can be used in conjunction with tight repeat - fail loops where it is desired that the rest of Windows can still operate in the background.

See Also

timer/2
wbusy/1
wflag/1

warea/5

get or check client area size and position

warea(*Window, Left, Top, Width, Height*)

+Window	<window_handle>
?Left	<integer> or <variable>
?Top	<integer> or <variable>
?Width	<integer> or <variable>
?Height	<integer> or <variable>

Get or set the *Left*, *Top* corner coordinates, and the *Width* and *Height* dimensions, of the given *Window*'s client area. All numerical values are in pixel units, where (0,0) is the left top of the screen.

This predicate cannot be used to resize or reposition a window (see *wsize/5*), but can be used to help align other GUI items within the client area of the given window.

See Also

wclass/2
wclose/1
wccreate/8
wcreate/8
wdcreate/7
wdict/1
wenable/2
wfind/3
wfocus/1
wlink/3
wndhdl/2
wshow/2
wsize/5
wstyle/2
wtcreate/6
wtext/2
wucreate/6

wbclose/1

close a bitmap

wbclose(*Bitmap*)

+Bitmap <atom>

Close the named *Bitmap*, returning its resources to Windows.

See Also
wbdict/1
wbload/2
wbopen/2

wbdict/1

return a dictionary of bitmaps

wbdict(*Bitmaps*)

+Bitmaps <variable>

Return a list of all the currently defined bitmaps (*Bitmaps*).

See Also
wbclose/1
wbload/2
wbopen/2

wbload/2

load a bitmap from a disk file

wbload(*Bitmap, File*)

+Bitmap	<atom>
+File	<atom>

Open the named *Bitmap* from the specified disk *File*.

See Also
wbclose/1
wbdict/1
wbopen/2

wbopen/2

load a bitmap from local resources

wbopen(*Bitmap, Name*)

+Bitmap	<atom>
+Name	<atom>

Open the named resource (*Name*) from the LPA Prolog resource file using the logical name (*Bitmap*).

See Also
wbclose/1
wbdict/1
wbload/2

wbtnsel/2

get or set selection state of a button

wbtnsel(*Window, Status*)

+Window	<window_handle>
?Status	<integer> or <variable>

Get or set the selection status of the given radio or checkbox button. The *Window* argument is the handle of the button. The *Status* argument is a button status value.

See Also
wccreate/8
wcreate/8
wdcreate/7

wbusy/1

get or set the busy cursor flag

wbusy(*CursorFlag*)

?CursorFlag	<integer> in the range {0..12}, <atom> or <variable>

Set or test the Windows busy cursor status. The *CursorFlag* argument may be a flag value or a variable. If the value of the *CursorFlag* argument is 1 or greater then the busy cursor flag is set.

Note that when the busy cursor is set, the mouse cannot be used for any purpose, and menus and dialogs are effectively disabled. This flag also prevents switching to other applications, so care must be taken in its use.

There are a number of predefined window cursors available for use with the *wbusy/1* predicate. The predicate *wcopen/2* can be used to load additional cursors from local resources.

See Also
timer/2
wait/1
wflag/1

wcclose/1

close a cursor

wcclose(*Cursor*)

+Cursor <atom>

Close the named *Cursor*, returning its resources to Windows.

See Also
wcdict/1
wcopen/2

wccreate/8

create a control window

wccreate(*Window*, *Class*, *Title*, *Left, Top, Width, Height, Style*)

+Window	<window_handle>
+Class	<atom>
+Title	<string>
+Left	<integer>
+Top	<integer>
+Width	<integer>
+Height	<integer>
+Style	<list>

Create a control *Window* with the given C*lass*, *Title*, *Left* - *Top* corner coordinates, *Width* - *Height* dimensions, and *Style*. The *Window* argument is of the form (*Parent*,*ID*), where *Parent* is the handle of a top-level window and *ID* is the handle of the control window. The *Class* argument is one of the predefined control window classes. The *Style* argument is a list of logical window styles which are combined to create the 32-bit integer which is passed directly to Windows. This predicate can combine any of the generic window styles with the styles for the given class. Note: you should always include the ws_child style in the *Style* list.

See Also

warea/5
wclass/2
wclose/1
wcreate/8
wdcreate/7
wdict/1
wenable/2
wfind/3
wfocus/1
wlink/3
wndhdl/2
wshow/2
wsize/5
wstyle/2
wtcreate/6
wtext/2
wucreate/6

wcdict/1

return a dictionary of cursors

wcdict(*Cursors*)

+Cursors <variable>

Return a list, *Cursors*, of all the currently defined cursors.

See Also
wcclose/1
wccreate/8
wcopen/2

wclass/2

check or get the class of a given window

wclass(*Window*, *Class*)

+Window	<window_handle>
?Class	<variable> or <atom>

Checks or returns the *Class* of the given *Window*. When looking at windows created by LPA Prolog for Windows, the *Class* argument may be an atom or a variable in which case it will be bound to a window class name. When looking at windows created elsewhere any class may be returned.

Note: dialog windows are assigned a class by Windows itself and this class is returned to LPA Prolog undefined.

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wcreate/8
wdcreate/7
wdict/1
wenable/2
wfind/3
wfocus/1
wlink/3
wndhdl/2
wshow/2
wsize/5
wstyle/2
wtcreate/6
wtext/2
wucreate/6

wclose/1

close a window

wclose(*Window*)

+Window	<window_handle>

Closes the named window, freeing up the memory resources it occupies. Note that when a window is closed all its children are closed automatically. Up to 256 top-level windows may be open at any one time, but this depends upon the availability of sufficient system memory resources, so it may be necessary to close one in order to open another.

The contents of a window are lost when it is closed, so you should save any such contents should you wish to use these data at a later time. See *wedtsel/3* and *wedttxt/2* for more information about retrieving text from "edit" or "editor" control windows.

See Also

warea/5
wclass/2
wccreate/8
wcreate/8
wdcreate/7
wdict/1
wenable/2
wfind/3
wfocus/1
wlink/3
wndhdl/2
wshow/2
wsize/5
wstyle/2
wtcreate/6
wtext/2
wucreate/6

wcopen/2

load a cursor from local resources

wcopen(*Cursor, ResourceName*)

+Cursor	<atom>
+ResourceName	<atom>

Open the named *Cursor* with the given *ResourceName* from the LPA Prolog resource file. The *Cursor* name can then be used whenever the cursor is needed.

See Also
wcclose/1
wccreate/8
wcdict/1

wcount/4

get char, word and line counts for the given window

wcount(*Window*, *Characters*, *Words*, *Rows*)

+Window	<window_handle>
-Characters	<variable>
-Words	<variable>
-Rows	<variable>

Returns the number of *Characters*, *Words* and *Rows* in the given *Window*. No side effects are caused by this predicate, which is used for the gathering of information only.

See Also
wtext/2

wcreate/8

create a window

wcreate(*Window*, *Class*, *Title*, *Left, Top, Width, Height, Style*)

+Window	<window_handle>
+Class	<atom>
+Title	<string>
+Left	<integer>
+Top	<integer>
+Width	<integer>
+Height	<integer>
+Style	<integer>

The predicate *wcreate/8* creates a window with the given *Window* name, *Class*, *Title*, *Left* - *Top* corner coordinates, *Width* - *Height* dimensions and *Style*. The window will be created in one of several styles, depending upon the given handle and style: if the handle is an atom, a top level window is created; if the handle is a conjunction of the form (window,id), then a control window with the given ID is created within the given window. The Class argument defines the type of window to be created; it may be either a predefined Windows class or an LPA defined window class.

The *Style* argument is a 32-bit integer that specifies an available window style which is passed directly to Windows. For top level windows (or MDI children), the text argument forms the window title (style permitting) of the window; for control windows, the text is the label of the control (where appropriate).

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wdcreate/7
wdict/1
wenable/2
wfind/3
wfocus/1
wlink/3
wndhdl/2
wshow/2
wsize/5
wstyle/2
wtcreate/6
wtext/2
wucreate/6

wdcreate/7

create a dialog window

wdcreate(*Window*, *Title*, *Left, Top, Width, Height, Style*)

+Window	<window_handle>
+Title	<string>
+Left	<integer>
+Top	<integer>
+Width	<integer>
+Height	<integer>
+Style	<list>

Create a dialog window with the given *Window*, *Title*, *Left - Top* corner, *Width - Height* dimensions, and *Style*. The *Window* argument must be an atom. *Style* is a list of logical window styles which are combined to create the 32-bit integer which is passed directly to Windows. This predicate can only use the generic window styles. Note: at present all dialogs must include the style 'ws_popup' to allow the dialog to function correctly stand-alone.

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wcreate/8
wdict/1
wenable/2
wfind/3
wfocus/1
wlink/3
wndhdl/2
wshow/2
wsize/5
wstyle/2
wtcreate/6
wtext/2
wucreate/6

wdict/1

get all currently open windows

wdict(*Wdict*)

-*Wdict* <variable>

Returns the window dictionary. If *Windows* is a variable it becomes instantiated to the list of user-defined windows. When a window is created, its name is added automatically to the window dictionary. It is removed when the window is closed.

Returns the dictionary of all currently defined "text", "user" and "dialog" type windows as a list *Wdict*.

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wcreate/8
wdcreate/7
wenable/2
wfind/3
wfocus/1
wlink/3
wndhdl/2
wshow/2
wsize/5
wstyle/2
wtcreate/6
wtext/2
wucreate/6

wedtclp/2

perform a clipboard function

wedtclp(*Window*, *Function*)

+Window	<window_handle>
+Function	<integer> in the domain {-5,-4,-3,2,1,1,2,3,4,5}

Perform the specified clipboard function on the given edit window. The *Window* argument is a window handle. The *Function* argument is a clipboard function value.

See Also
wedtfnd/6
wedtlin/4
wedtpxy/4
wedtsel/3
wedttxt/2

wedtfnd/6

find a text string in an "edit" or "editor" control window

wedtfnd(*Window*, *Start*, *End*, *String*, *StartMatch*, *EndMatch*)

+Window	<window_handle>
+Start	<integer> in the range [0..32767]
+End	<integer> in the range [0..32767]
+String	<string>
-StartMatch	<variable>
-EndMatch	<variable>

Search the given "edit" or "editor" control *Window* for the given text *String* within the given *Start* and *End* points. The start and finish of the first matching string is returned as a pair of integers, *StartMatch* and *EndMatch*. As a special case, the search text may be specified as an empty string. In this case, the start and finish of the next space-delimited token is returned. No side effects are caused by this predicate, which is used for the gathering of information only. The returned parameters may be passed directly into *wedtsel/3* if it is desired to move the selection to the found string.

See Also
wedtclp/2
wedtlin/4
wedtpxy/4
wedtsel/3
wedttxt/2

wedtlin/4

get offsets a line in an "edit" or "editor" control window

wedtlin(*Window*, *Offset*, *Start*, *Finish*)

+Window	<window_handle>
+Offset	<integer> in the range [0..32767]
?Start	<variable> or <integer>
?Finish	<variable> or <integer>

Returns the *Start* and *Finish* of the line of text containing the given character offset in the given "edit" or "editor" control *Window*, or tests the given values for correctness. The offsets returned include everything on the given line, but not the carriage return/line feed. No side effects are caused by this predicate, which is used for the gathering of information only.

See Also
wedtclp/2
wedtfnd/6
wedtpxy/4
wedtsel/3
wedttxt/2

wedtpxy/4

convert between linear offset and x, y coordinates in "edit" or "editor" windows

wedtpxy(*Window*, *Offset*, *X, Y*)

+Window	<window_handle>
?Offset	<variable> or <integer>
?X	<variable> or <integer>
?Y	<variable> or <integer>

Returns the *X* and *Y* coordinates that are the equivalent of a given character *Offset*, or returns the character *Offset* of the given *X* and *Y* values, or tests the given values for correctness. The values are computed for the given "edit" or "editor" *Window*. No side effects are caused by this predicate, which is used for the gathering of information only.

See Also
wedtclp/2
wedtfnd/6
wedtlin/4
wedtsel/3
wedttxt/2

wedtsel/3

get or set selection in an "edit" or "editor" control window

wedtsel(*Window*, *First*, *Second*)

?Window	<window_handle>
?First	<variable> or <integer>
?Second	<variable> or <integer>

Sets the text selection area in the given "edit" or "editor" control *Window* to start and finish at the given *First* and *Second* values, or returns the existing values. This predicate causes a direct side effect on the window, whose cursor moves to the position specified.

Note that the start and finish positions can be given in either order: the flashing caret is positioned at the end specified by the *First* value. Windows does not provide the caret position when retrieving the selection: the smallest value is always returned in the *First* parameter.

See Also
wedtclp/2
wedtfnd/6
wedtlin/4
wedtpxy/4
wedttxt/2

wedttxt/2

get or set the text of the given "edit" or "editor" window

wedttxt(*Window*, *Text*)

+Window	<window_handle>
?Text	<string> or <variable>

Gets or sets the *Text* in the selected area of the given "edit" or "editor" control *Window*. This predicate causes a direct side effect on the window, whose text contents and selection area may be changed as specified.

Although it is not documented in the Windows SDK, it would appear that MDI "text" windows and "editor" control windows used by LPA Prolog have a maximum size of exactly 30000 characters. See *wcount/4* and *wedtsel/3* for information about how to check, prior to inserting text into an "editor" window, whether the insertion is likely to prove successful.

See Also

wedtclp/2
wedtfnd/6
wedtlin/4
wedtpxy/4
wedtsel/3
wedttxt/2

wenable/2

get or set window enable status

wenable(*Window*, *Status*)

+Window	<window_handle>
?Status	<variable> or <integer> in the domain {0,1}

Get or set the enable status of the given window. The *Window* argument is a window handle. The *Status* argument is a window status value.

When you set the *Status* of a button to 0 the button is made grey and cannot be selected; setting the *Status* to 1 gives the button its default appearance and the button may now be selected. For other windows the effect of setting the *Status* is not visible; the window can simply no longer be selected.

When you set the status of LPA Prolog's main window, this may also affect the change, find and status boxes. If you disable the main window, any of the boxes which are enabled become temporarily disabled. If you then reenable the main window the temporarily disabled windows are also reenabled. Any boxes that are already disabled remain unaffected by disabling and subsequently reenabling the main window.

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wcreate/8
wdcreate/7
wdict/1
wfind/3
wfocus/1
wlink/3
wndhdl/2
wshow/2
wsize/5
wstyle/2
wtcreate/6
wtext/2
wucreate/6

wfclose/1

close a font

wfclose(*Font*)

+Font <atom>

Close the named *Font*, returning its resources to Windows. You should not close a font which is still in use in one or more windows. Fonts are global Windows resources and are not automatically freed when their creating application terminates. It is important to close fonts explicitly, where possible, before terminating a LPA Prolog session. See *wfcreate/4*, *wfdict/1* and *fonts/1* for more details on creating and maintaining fonts.

See Also

ansoem/2
fonts/1
wfcreate/4
wfdata/5
wfdict/1
wfont/2
wfsize/4

wfcreate/4

create a font

wfcreate(*Font*, *Typeface*, *Size*, *Style*)

+Font	<atom>
+Typeface	<atom>
+Size	<integer>
+Style	<integer>

Create the named font, using the given typeface, size and style. The *Font* argument is an atom that you can use in your programs to refer to the font. After the font has been created this is the name that appears on the font dictionary (see *wfdict/1*). The *Typeface* argument is an atom which is the name of a currently available typeface, these can be found using the predicate *fonts/1*. The *Size* argument is given in units of 1 pt (approx 1/72 inch). The *Style* argument is a font style value.

Note that any existing font with the same name will be closed automatically before the new one is created: you should not close a font which is still in use in one or more windows.

See Also

ansoem/2
fonts/1
wfclose/1
wfdata/5
wfdict/1
wfont/2
wfsize/4

wfdata/5

check or get the typeface, size, style and ascent of the given logical font

wfdata(*Font*, *TypeFace*, *Size*, *Style*, *Ascent*)

+Font	<atom> or <integer> in the domain {0,1}
?TypeFace	<atom> or <variable>
?Size	<integer> or <variable>
?Style	<integer> in the domain {0,1,2,3} or <variable>
?Ascent	<integer> or <variable>

Check or get the *TypeFace*, *Size* and *Style* and *Ascent* of the given logical *Font*. *Font* should be the atom name of a currently defined logical font, created using *wfcreate/4*. *TypeFace* should be the atom name of a currently defined Windows system typeface or a variable. *Size* should be the size of the font given in logical units or a variable. *Style* should be a font style value or a variable. *Ascent* should be an integer or a variable and corresponds to the distance in logical units between the base line of the character and the top of the character cell.

Note that 1 logical unit is usually equivalent to 1 point given the default mapping mode of 1:1.

Sometimes when you create a font, the specified typeface may not actually be a currently defined Windows system typeface, and in this case Windows assigns a substitute. The *wfdata/5* predicate can also be used to check whether a call to *wfcreate/4* has defined a font with the specified typeface.

See Also

ansoem/2
fonts/1
wfclose/1
wfcreate/4
wfdict/1
wfont/2
wfsize/4

wfdict/1

return a dictionary of fonts

wfdict(*Wfdict*)

-*Wfdict* <variable>

Return the dictionary of all currently defined fonts as a list, *Wfdict*. See *wfcreate/4*, *wfclose/1* and *fonts/1* for more details on creating and maintaining fonts.

See Also
ansoem/2
fonts/1
wfclose/1
wfcreate/4
wfdata/5
wfont/2
wfsize/4

wfind/3

find the handle for a named window

wfind(*Class*, *Title*, *Handle*)

+Class	<atom>
+Title	<string>
-Handle	<variable>

Find the *Handle* of the first top level window whose *Class* and *Title* are given. If the class is given as an empty atom, all classes are searched; similarly, if the title is given as an empty string, all windows within the given class are searched. *Title* should have the same type as *Class*.

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wcreate/8
wdcreate/7
wdict/1
wenable/2
wfocus/1
wlink/3
wndhdl/2
wshow/2
wsize/5
wstyle/2
wtcreate/6
wtext/2
wucreate/6

wflag/1

get or set the Windows message interrupt flag

wflag(*Flag*)

?Flag	<variable> or <integer> in the domain {0,1}

Get or set the state of the Windows message interrupt flag. When clear, this flag suppresses the reporting of messages to Prolog programs, disabling all user-defined message, menu, DLL or dialog handlers; when set, such messages cause the temporary interruption of Prolog's execution, passing control to any user-defined handlers. The *Flag* argument is an interrupt state value.

Note that the interrupt flag is a one-shot flag, which needs explicit reenabling after a message has been handled. Up to 256 messages may be queued at any one time, but each time the flag is set, at most one message will be signalled to the message handler.

It is essential that the Windows message interrupt flag be set at all times other than when processing an interrupt to ensure the correct operation of LPA Prolog's menus.

See Also
timer/2
wait/1
wbusy/1

wfocus/1

get or set input focus to a window

wfocus(*Window*)

?Window	<window_handle>

Sets input focus to the named window, activating it and bringing it to the top of the display stack. If the argument is given as a variable, the name of the window currently in focus is returned if possible.

Note: any window can be given focus, not just named windows. When setting focus to a dialog or MDI child window, it is best to explicitly identify the control which is to receive focus, and not the top level window itself.

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wcreate/8
wdcreate/7
wdict/1
wenable/2
wfind/3
wlink/3
wndhdl/2
wshow/2
wsize/5
wstyle/2
wtcreate/6
wtext/2
wucreate/6

wfont/2

get or set the font of a window

wfont(*Window*, *Font*)

+Window	<window_handle>
+Font	<atom> or <integer> in the domain {0,1}

Set the font for the given window. The *Font* argument is either a variable, a named font (see *wfcreate/4*) or a special numeric font. If the *Font* argument is a variable it is bound to the current font of the window if one has been set previously.

See Also

ansoem/2
fonts/1
wfclose/1
wfcreate/4
wfdata/5
wfdict/1
wfsize/4

wfsize/4

check or get the height and width of the given string in the given font

wfsize(*Font*, *String*, *Width*, *Height*)

+Font	<atom> or <integer> in the domain {0,1}
+String	<string>
?Width	<variable> or <integer>
?Height	<variable> or <integer>

Check or return the *Width* and *Height* of the given *String* using the given logical *Font*. *Height* is the height of the character cells in pixels, which is constant for any character font. *Width* is the distance, in pixels, from the start of the first character cell to the end of the last character cell in the *String*. Note that this distance may not neccessarily be the sum of all the character cells in the string, as the character cells in some fonts overlap.

See Also

ansoem/2
fonts/1
wfclose/1
wfcreate/4
wfdata/5
wfdict/1
wfont/2

wgfx/2

perform a windows graphics sequence

wgfx(*Win, Glist*)

+Win	<window_handle>
+Glist	<list_of <compound_term> >

Perform a sequence of graphics instructions in the given window. The *Glist* argument must be a list of valid graphics functions.

See Also

wgfx/6
wgfxadd/5
wgfxcur/2
wgfxget/5
wgfxmap/5
wgfxorg/3
wgfxpnt/1
wgfxsub/5
wgfxtst/5

wgfx/6

perform a clipped windows graphics sequence

wgfx(*Win, Glist, X, Y, X1, Y1*)

+Win	<window_handle>
+Glist	<list_of <compound_term> >
+X	<integer>
+Y	<integer>
+X1	<integer>
+Y1	<integer>

Perform a list of graphics instructions in the given window using the given clip region. The *Win* argument is the handle of a window created by Prolog. The *GList* argument is a list of valid graphics instructions. The arguments *X*, *Y*, *X1* and *Y1* are all integers that specify the device coordinates for the clipping region.

See Also

wgfx/2
wgfxadd/5
wgfxcur/2
wgfxget/5
wgfxmap/5
wgfxorg/3
wgfxpnt/1
wgfxsub/5
wgfxtst/5

wgfxadd/5

add rectangle to graphics update region

wgfxadd(*Win, Left, Top, Right, Bottom*)

+Win	<window_handle>
+Left	<integer>
+Top	<integer>
+Right	<integer>
+Bottom	<integer>

Add the rectangle specified by *Left*, *Top*, *Right* and *Bottom* parameters to the update rectangle for the given "grafix" window (*Win*). The resultant graphics update region will be a rectangle that bounds both the previous update region and the rectangle to be added.

See Also

wgfx/2
wgfx/6
wgfxcur/2
wgfxget/5
wgfxmap/5
wgfxorg/3
wgfxpnt/1
wgfxsub/5
wgfxtst/5

wgfxcur/2

get or set the cursor for a grafix window

wgfxcur(*Win, Cursor*)

+Win	<window_handle>
+Cursor	<atom>

Get or set the type of cursor for the given "grafix" window. The *Cursor* argument may either be an atom or a predefined cursor value. If the *Cursor* argument is an atom it should refer to a cursor previously opened using *wcopen/2*.

See Also

wgfx/2
wgfx/6
wgfxadd/5
wgfxget/5
wgfxmap/5
wgfxorg/3
wgfxpnt/1
wgfxsub/5
wgfxtst/5

wgfxget/5

get the graphics update region

wgfxget(*Win, Left, Top, Right, Bottom*)

+Win	<window_handle>
?Left	<variable> or <integer>
?Top	<variable> or <integer>
?Right	<variable> or <integer>
?Bottom	<variable> or <integer>

Get the update rectangle for the given "grafix" window (*Win*).

See Also

wgfx/2
wgfx/6
wgfxadd/5
wgfxcur/2
wgfxmap/5
wgfxorg/3
wgfxpnt/1
wgfxsub/5
wgfxtst/5

wgfxmap/5

get or set the graphics mapping

wgfxmap(*Win, XLogical, YLogical, XPhysical, YPhysical*)

+Win	<window_handle>
?XLogical	<variable> or <integer>
?YLogical	<variable> or <integer>
?XPhysical	<variable> or <integer>
?YPhysical	<variable> or <integer>

Get or set the ratio between the logical coordinates (*XLogical* and *YLogical*) and the physical coordinates (*XPhysical and YPhysical*) for the given "grafix" window (*Win*), invalidating the window if necessary.

Warning: The Windows GDI module contains some bugs in its internal parameter validation routines, and these can manifest themselves as General Protection Faults (GPFs) when certain graphics operations are carried out. In particular, the plotting of ellipses is known to cause problems when their target window has a very high (magnified) scaling factor. There is, unfortunately, no way in which to prejudge which calls are going to cause problems, so to avoid them, please observe the following precautions:

Resist using wgfxmap/5 to enlarge the scaling of a graphics window by more than a ratio of about 8:1,

or:

Avoid plotting ellipses, polygons or rounded rectangles on windows whose scaling factor exceeds this value.

See Also

wgfx/2
wgfx/6
wgfxadd/5
wgfxcur/2
wgfxget/5
wgfxorg/3
wgfxpnt/1
wgfxsub/5
wgfxtst/5

wgfxorg/3

get or set the graphics origin for a given window

wgfxorg(*Win, XView, YView*)

+Win	<window_handle>
?XView	<variable> or <integer>
?YView	<variable> or <integer>

Get or set the *XView* and *YView* viewport origins for the given "grafix" window (*Win*). Scroll the window if necessary.

See Also

wgfx/2
wgfx/6
wgfxadd/5
wgfxcur/2
wgfxget/5
wgfxmap/5
wgfxpnt/1
wgfxsub/5
wgfxtst/5

wgfxpnt/1

force the painting of the graphics update region

wgfxpnt(*Win*)

+Win	<window_handle>

Clear the current graphics update region for the given "grafix" window (*Win*) and send the *msg_paint* message to LPA Prolog.

See Also

wgfx/2
wgfx/6
wgfxadd/5
wgfxcur/2
wgfxget/5
wgfxmap/5
wgfxorg/3
wgfxsub/5
wgfxtst/5

wgfxsub/5

subtract a rectangle from a graphics update region

wgfxsub(*Win, Left, Top, Right, Bottom*)

+Win	<window_handle>
+Left	<integer>
+Top	<integer>
+Right	<integer>
+Bottom	<integer>

Subtract the rectangle specified by the *Left*, *Top*, *Right* and *Bottom* parameters from the update rectangle for the given "grafix" window (*Win*).

See Also

wgfx/2
wgfx/6
wgfxadd/5
wgfxcur/2
wgfxget/5
wgfxmap/5
wgfxorg/3
wgfxpnt/1
wgfxtst/5

wgfxtst/5

perform a windows graphics hit test

wgfxtst(*Win, GList, X, Y, Hits*)

+Win	<window_handle>
+GList	<list_of <compound_term> >
+X	<integer>
+Y	<integer>
?Hits	<variable> or <integer>

Perform a list of virtual graphics instructions in the given window and check or get the number of solid objects that coincide with the device x and y coordinates. The *GList* argument is a list that may contain any of the *wgfx/2* graphics instructions, but there are two sub-groups which have some effect. These sub-groups are: graphics instructions used to detect hits with wgfxtst/5 and graphics instructions that affect the way in which hits are reported by *wgfxtst/5*. The remaining *wgfx/2* functions are ignored during testing.

See Also

wgfx/2
wgfx/6
wgfxadd/5
wgfxcur/2
wgfxget/5
wgfxmap/5
wgfxorg/3
wgfxpnt/1
wgfxsub/5

wiclose/1

close an icon

wiclose(*Icon*)

+Icon <atom>

Close the named *Icon*, returning its resources to Windows, where *Icon* was previously opened using *wiopen/2* or *wiload/3*.

See Also
widict/1
wiload/3
wiopen/2

widict/1

return a dictionary of icons

widict(*IDict*)

-IDict <variable>

Return the dictionary, *IDict*, of all currently defined icons as a list.

See Also
wiclose/1
wiload/3
wiopen/2

wiload/3

load an icon from a disk file

wiload(*Icon, File, Index*)

+Icon	<atom>
+File	<atom>
+Index	<integer>

Open the named *Icon* from the specified disk *File* and icon *Index*.

See Also
wiclose/1
widict/1
wiopen/2

winapi/5

call a C function defined in the Windows API environment or in a DLL

winapi(*Module*,*Function*,*Inputs*,*Result*,*Outputs*)

+Module	<atom>
+Function	(<atom>,<atom>)
+Inputs	<list>
-Result	<variable>
-Outputs	<variable>

Call a C function in the given module with the specified input arguments and return the result and output arguments. The *Module* argument is an atom that is the name of a module. The *Function* argument is a conjunction of two atoms: the first atom is a function type cast and the second is the name of the function to be called.

The *Inputs* argument is a list containing typed input parameters to the given function. Each element of the *Inputs* list is a parameter type cast. A parameter type cast may be a simple conjunction of a data type and its associated data or a list (in which case it indicates a structure).

The members of a structure list are structure element type casts. Structure element type casts may only be simple conjunctions of data types and their associated data. Structures cannot be nested.

The *Result* argument is a variable that becomes bound to the result of the given function. If a text function returns a non-text result (such as "TRUE", "FALSE" or "NULL") the *Result* argument is bound to the appropriate integer instead of a string.

The *Outputs* argument is a variable that becomes bound to the output parameters of the given function, these will match exactly the types specified in the *Inputs* argument. If a text parameter or structure element specified in the *Inputs* list comes back as a non-text result (such as "TRUE", "FALSE" or "NULL") the corresponding element in the *Outputs* argument is bound to the appropriate integer instead of a string.

Note: Microsoft Windows itself does not provide a safe means of argument type checking in API or DLL function calls, so incorrect use of the *winapi/5* predicate can cause application errors and general protection faults: care must therefore be taken to ensure the argument type definitions provided in the call match those demanded by the Windows API or DLL function. It is also important to call only functions defined as "FAR PASCAL".

See Also

dll_hook/3
lcall/4
lclose/1
ldict/1
lopen/1
'?DLL?'/3

window_handler/2

get or set the current message handler for the given window

window_handler(*Window*, *Handler*)

+Window	<window_handle>
?Handler	<atom> or <variable>

Attach the given message handler to the given window. The *Window* argument should be the handle of a Prolog generated window. If the *Handler* argument is an atom this should be the functor of a message handler, with four arguments, that will handle messages from the *Window* (or its child windows). The handler for a dialog should be a Prolog program of the form:

```
Handler(MessageWindow, Message, Data, HandleResult) :-
    Body
```

MessageWindow is of type <window_handle> and is the window that generated the message. *Message* is the text equivalent of the message number that was generated. *Data* is any data associated with the message (such as the (X,Y) mouse coordinates for the *msg_mousemove* message in graphics windows). If the *HandleResult* argument is bound then *MessageWindow* (or its parent in the case of child windows) will be hidden. If the window is being called by *call_dialog/2* then the bound *HandleResult* will be returned as the second argument of this call.

If the *Handler* argument is a variable the handler currently set for the window is returned. By default all windows that are created are handled by the built-in *window_handler/4*. It is possible to use this default message handler by making a call to it in your own handler programs. To effectively remove a handler from a window simply set the handler for the window to be "window_handler", as in the following call:

```
?- window_handler(my_window,window_handler).
```

See Also

call_dialog/2
window_handler/4
show_dialog/1
wdcreate/7

window_handler/4

system defined handler for windows

window_handler(*Win, Msg, Data, Result*)

+Win	<window_handle>
+Msg	<atom>
+Data	<integer>
+Result	<variable>

Invoke the default behaviour for handling the given window. The *Win* argument is a Prolog window handle. The *Msg* argument is an atom that should be a message name. The *Data* argument should be any valid data associated with the message. The *Result* argument should be a variable, which may get bound as a result of handling the message.

If the message is one of type msg_button the *Result* argument will be bound to the text of the button that sent the message. If the message is one of type msg_close the *Result* argument will be bound to the atom close.

See Also

call_dialog/2
window_handler/2
show_dialog/1
wdcreate/7

wiopen/2

load an icon from local resources

wiopen(*Icon, IconRes*)

+Icon	<atom>
+IconRes	<atom>

Open the named icon resource (*IconRes*) from the LPA Prolog resource file using the logical name (*Icon*).

See Also
wiclose/1
widict/1
wiload/3

wlbxadd/3

add an item to a list box

wlbxadd(*Window*, *Position*, *String*)

+Window	<window_handle>
+Position	<integer>
+String	<string>

Add a *String* to the "listbox" control *Window* at the given *Position*. If the position is given as -1, the item is inserted or appended to the list box depending upon the list box style. Entries in a listbox are numbered from 0.

See Also

wlbxdel/2
wlbxfnd/4
wlbxget/3
wlbxsel/3

wlbxdel/2

delete an item from a list box

wlbxdel(*Window*, *Position*)

+Window	<window_handle>
+Position	<integer>

Delete the item at the given *Position* in the given "listbox" control *Window*. Entries in a listbox are numbered from 0.

See Also

wlbxadd/3
wlbxfnd/4
wlbxget/3
wlbxsel/3

wlbxfnd/4

find a string in a list box

wlbxfnd(*Window*, *Start*, *String*, *Position*)

+Window	<window_handle>
+Start	<integer>
+String	<string>
-Position	<variable>

Return the *Position* of a partial match *String* in the given "listbox" control *Window*, starting search one place after the given *Start*. Entries in a listbox are numbered from 0. If *String* is the empty string `` then *wlbxfnd/4* will return the *Position* of the entry following the *Start*.

See Also

wlbxadd/3
wlbxdel/2
wlbxget/3
wlbxsel/3

wlbxget/3

get an item from a list box

wlbxget(*Window*, *Position*, *String*)

+Window	<window_handle>
+Position	<integer>
-String	<variable>

Get the *String* at the given *Position* in the given "listbox" control *Window*. Entries in a listbox are numbered from 0. This predicate will fail if the listbox has a number of entries that is less than or equal to the given position.

See Also

wlbxadd/3
wlbxdel/2
wlbxfnd/4
wlbxsel/3

wlbxsel/3

get or set selection in a list box

wlbxsel(*Window*, *Position*, *State*)

+Window	<window_handle>
+Position	<integer>
?State	<integer> or <variable>

Set or get the selection *State* of the item at the given *Position* in the given "listbox" control *Window*. If the *Position* is given as -1, and the listbox is a multi-choice list box, the selection state is applied to all items. Entries in a listbox are numbered from 0. The *State* argument is a variable or a listbox selection state value.

See Also

wlbxadd/3
wlbxdel/2
wlbxfnd/4
wlbxget/3

wlink/3

find the handle for a linked window

wlink(*Window*, *Relation*, *Handle*)

+Window	<window_handle>
+Relation	<integer>
-Handle	<variable>

Find the *Handle* of a window related by a given *Relation* to the given *Window*. The *Relation* argument is window relation value.

Windows are defined in a hierarchical fashion using the notions of parents, children, siblings and owners. For example given a dialog that has a number of control fields, in Windows those control fields are said to be *children* of the main dialog window, the main dialog window is their *parent* and the children are all *siblings* whose order is dependent on the order in which they are defined.

There are subtle differences between 'parent' windows and 'owner' windows, which are beyond the scope of this manual. For more details please consult your Windows SDK documentation.

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wcreate/8
wdcreate/7
wdict/1
wenable/2
wfind/3
wfocus/1
wndhdl/2
wshow/2
wsize/5
wstyle/2
wtcreate/6
wtext/2
wucreate/6

wmclose/1

close a menu

wmclose(*Menu*)

+Menu <atom>

Close the named *Menu*, returning its resources to Windows. Note that you should not close a menu which is still in use in one or more windows. Menus are global Windows resources and are not automatically freed when their creating application terminates. It is important to close menus explicitly, where possible, before terminating a LPA Prolog session.

See Also

wmcreate/1
wmdict/1
wmnuadd/4
wmnudel/2
wmnuget/4
wmnunbl/3
wmnusel/3

wmcreate/1

create a menu

wmcreate(*Menu*)

+Menu <atom>

Create the named logical *Menu*. *Menu* should be an atom which will be used from then on whenever you want to perform an action on the *Menu*. These actions include: closing the menu, getting items from the menu, adding and deleting items, setting the selection state of items (ticked or not ticked) and setting the enable state of items. Note that any existing menu with the same name will be closed automatically before the new one is created: you should not close a menu which is still in use in one or more windows. Menus are global Windows resources and are not automatically freed when their creating application terminates. It is important to close menus explicitly, where possible, before terminating an LPA Prolog session.

See Also

wmclose/1
wmdict/1
wmnuadd/4
wmnudel/2
wmnuget/4
wmnunbl/3
wmnusel/3

wmdict/1

return a dictionary of menus

wmdict(*Menus*)

-Menus <variable>

Return a list of the currently defined *Menus*. Each menu name is the atom that was used when the menu was created using *wmcreate*/1.

See Also
wmclose/1
wmcreate/1
wmnuadd/4
wmnudel/2
wmnuget/4
wmnunbl/3
wmnusel/3

wmnuadd/4

add an item to a menu

wmnuadd(*Menu*, *Position*, *String*, *Item*)

+Menu	<atom> or 0
+Position	<integer>
+String	<string>
+Item	<integer> or <atom>

Add an *Item* and associated *String* to the given *Menu* at *Position*. If the *Position* is given as -1, the *Item* is appended to the *Menu*. *Menu* should be the name of a menu, or 0 (the Prolog menu bar). The *Item* may be an integer or an atom. If it is an integer, then it may either be 0, which specifies a separator entry (the *String* is ignored in this case), or a positive integer which will be generated as a message whenever the item is selected. If the *Item* is an atom, this is taken to be the handle of a sub-menu to be attached to *Menu*. Note that Prolog reserves the message values under 1,000 for internal use, and Windows itself reserves message values which are greater than around 64,000. Any menu items that you define should use messages between 1000 and 64000.

To add your own menu to the Prolog menu bar *Menu* should be 0. The message *msg_menu* is sent, whenever an item is selected from a menu, to the window currently in focus. A data parameter is also sent with this message and this is the ID for the menu item selected. To handle these messages you should attach a handler to each window that needs to be aware of your menus and IDs.

Hot keys (keys that can be used to select a menu item directly) can be added to menu items using an ampersand "&" (this should appear in the *String* that defines the text of the menu item before the letter you want to use as a hot key). Separators (lines that cannot be selected, for separating items on a menu) can be added by giving the integer 0 as the *Item* value.

WARNING: Windows cannot handle circular menu definitions, and will crash if you use such structures. Be very careful not to define menus that either call themselves or call other menus which in turn call themselves, and so on.

See Also

wmclose/1
wmcreate/1
wmdict/1
wmnudel/2
wmnuget/4
wmnunbl/3
wmnusel/3

wmnudel/2

delete an item from a menu

wmnudel(*Menu*, *Position*)

+Menu	<atom> or 0
+Position	<integer>

Delete the item at the given *Position* in the given *Menu*. *Menu* should be the name of a logical menu, or 0 (the menu bar). Note that if the item being deleted is a sub-menu, neither that menu nor its contents are destroyed: it is simply removed from the parent menu.

See Also

wmclose/1
wmcreate/1
wmdict/1
wmnuadd/4
wmnuget/4
wmnunbl/3
wmnusel/3

wmnuget/4

get an item from a menu

wmnuget(*Menu*, *Position*, *String*, *Item*)

+Menu	<atom> or 0
+Position	<integer>
-String	<variable>
-Item	<variable>

Get the *Item* and associated *String* at the given *Position* in the given *Menu*. *Menu* should be the name of a logical menu, or 0 (the menu bar). The *Item* returned may be an integer or an atom. If it is an integer, then it may either be 0, which specifies a separator entry, or a positive value greater than 1000. This value will be generated as a message whenever the item is selected. If the *Item* is an atom then it is the handle of a sub-menu.

See Also

wmclose/1
wmcreate/1
wmdict/1
wmnuadd/4
wmnudel/2
wmnunbl/3
wmnusel/3

wmnunbl/3

get or set enable status of an item on a menu

wmnunbl(*Menu*, *Position*, *Status*)

+Menu	<atom> or 0
+Position	<integer>
?Status	<integer> or <variable>

Get or set the enable *Status* of the item at the given *Position* in the given *Menu*. The *Status* argument is either a variable or a menu enable status value.

Menu should be the name of a logical menu created with *wmcreate/1*, or 0 (the menu bar).

See Also

wmclose/1
wmcreate/1
wmdict/1
wmnuadd/4
wmnudel/2
wmnuget/4
wmnusel/3

wmnusel/3

get or set selection state of an item on a menu

wmnusel(*Menu*, *Position*, *State*)

+Menu	<atom> or 0
+Position	<integer>
?Status	<integer> or <variable>

Set or get the selection status of the item at the given position in the given menu. The *Status* argument is either a variable or a menu selection status value. The *Menu* argument should be the name of a logical menu created with *wmcreate/1*, or 0 (the menu bar).

See Also

wmclose/1
wmcreate/1
wmdict/1
wmnuadd/4
wmnudel/2
wmnuget/4
wmnunbl/3

wndhdl/2

convert between window and handle

wndhdl(*Window*, *RawHandle*)

?Window	<window_handle> or <variable>
?RawHandle	<integer> or <variable>

Convert between an LPA Prolog *Window* handle and raw window handle, *RawHandle*. Raw handles are 16-bit, unsigned integers, as used internally by Windows. LPA Prolog handles may also be in this form, but wherever possible, are given as symbolic names or (window,id) conjunctions. This predicate can be used to find the 16-bit handle of an LPA window so that it may be manipulated by an external process such as a DLL.

This predicate can also be used to test if a given integer is the handle of a LPA Prolog window. The *wndhdl/2* predicate succeeds if the integer is the handle of a currently open window and fails if it is not. This can be useful when handling msg_fuzzy messages, which may come sometimes after the window that generated the message has already been closed. In this case the window is indicated using its raw handle. A call to *wndhdl/2* could be made using that raw handle to determine if it refers to a currently open window.

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wcreate/8
wdcreate/7
wdict/1
wenable/2
wfind/3
wfocus/1
wlink/3
wshow/2
wsize/5
wstyle/2
wtcreate/6
wtext/2
wucreate/6

wprnend/1

finish or abort the use of the printer

wprnend(*Action*)

+Action <integer> in the domain {0,1}

Finish or abort the current use of the printer. The *Action* argument is a terminate code value. If during printing, the user clicks on a "Cancel" button, this predicate fails and the print job is aborted.

See Also
wprngfx/1
wprngfx/5
wprnini/4
wprnmap/4
wprnorg/2
wprnpag/1
wprnres/4
wprnstt/1

wprngfx/1

perform a printer graphics sequence

wprngfx(*GList*)

+GList	<list_of <compound_term> >

Perform a list of graphics commands on the printer. The *GList* argument is a list of valid graphics instructions.

See Also

wprnend/1
wprngfx/5
wprnini/4
wprnmap/4
wprnorg/2
wprnpag/1
wprnres/4
wprnstt/1

wprngfx/5

perform a clipped printer graphics sequence

wprngfx(*GList, X*, *Y*, *X1, Y1*)

+GList	<list_of <compound_term> >
+X	<integer>
+Y	<integer>
+X1	<integer>
+Y1	<integer>

Perform a list of graphics commands on the printer within the given clipping region. The *GList* argument is a list of valid graphics instructions. The arguments *X*, *Y*, *X1* and *Y1* are all integers that specify the device coordinates for the clipping region. The commands are sent to the printer but no actual printing occurs until a call is made to *wprnend/1*.

See Also

wprnend/1
wprngfx/1
wprnini/4
wprnmap/4
wprnorg/2
wprnpag/1
wprnres/4
wprnstt/1

wprnini/4

initialise the printer

wprnini(*Document,Printer,Driver,Port*)

+Document	<atom>
+Printer	<atom> or <variable>
+Driver	<atom> or <variable>
+Port	<atom> or <variable>

Initialise the named document for the given printer, device driver and output device. The *Document* argument is an atom that specifies a logical document for the printer. If the *Printer Driver* and *Port* arguments are all atoms then they are set as the printer, driver and port for the given document.

See Also

wprnend/1
wprngfx/1
wprngfx/5
wprnmap/4
wprnorg/2
wprnpag/1
wprnres/4
wprnstt/1

wprnmap/4

get or set the printer graphics mapping

wprnmap(*XLogical,YLogical,XPhysical,YPhysical*)

?XLogical	<variable> or <integer>
?YLogical	<variable> or <integer>
?XPhysical	<variable> or <integer>
?YPhysical	<variable> or <integer>

Get or set the ratio between the logical coordinates (*XLogical* and *YLogical*) and the physical coordinates (*XPhysical and YPhysical*) for the currently selected printer.

See Also

wprnend/1
wprngfx/1
wprngfx/5
wprnini/4
wprnorg/2
wprnpag/1
wprnres/4
wprnstt/1

wprnorg/2

get or set the printer graphics origin

wprnorg(*XView, YView*)

?XView	<variable> or <integer>
?YView	<variable> or <integer>

Get or set the *XView* and *YView* viewport origins for the currently selected printer.

See Also

wprnend/1
wprngfx/1
wprngfx/5
wprnini/4
wprnmap/4
wprnpag/1
wprnres/4
wprnstt/1

wprnpag/1

start a new printer page

wprnpag(*Page*)

-Page <variable>

Start printing on a new page, returning its page number. The commands to do this are sent to the printer but no actual printing occurs until a call is made to *wprnend/1*. If during printing, the user clicks on a "Cancel" button, this predicate fails and the print job is aborted.

See Also

wprnend/1
wprngfx/1
wprngfx/5
wprnini/4
wprnmap/4
wprnorg/2
wprnres/4
wprnstt/1

wprnres/4

get or check the printer resolution

wprnres(*HPagePix,VPagePix,HInchPix,VInchPix*)

?HPagePix	<integer> or <variable>
?VPagePix	<integer> or <variable>
?HInchPix	<integer> or <variable>
?VInchPix	<integer> or <variable>

Get or check the number of pixels per horizontal and vertical page, and per horizontal and vertical inch for the currently selected printer.

See Also

wprnend/1
wprngfx/1
wprngfx/5
wprnini/4
wprnmap/4
wprnorg/2
wprnpag/1
wprnstt/1

wprnstt/1

get or check the printer status

wprnstt(*Status*)

?Status <integer> in the range [0..3] or <variable>

Get or check the current status of the printer. The *Status* argument returns or checks a printer status value.

See Also

wprnend/1
wprngfx/1
wprngfx/5
wprnini/4
wprnmap/4
wprnorg/2
wprnpag/1
wprnres/4

wrange/4

get or set range of a scroll bar

wrange(*Window*, *Type*, *Lower*, *Upper*)

+Window	<window_handle>
+Type	<integer> in the domain {0,1,2}
+Lower	<integer>
+Upper	<integer>

Set the lower and upper limits of the given scroll bar type within the given window. Both the *Lower* and *Upper* arguments are signed 16-bit integers. The *Type* argument is a scrollbar type value. Scroll bars may be either connected to a window (i.e. given as part of the style definition of that window) or stand-alone (a separate control item of type "scrollbar"). Windows addresses these two types of scrollbar differently, so *wrange/4* needs to be provided with the appropriate type flag.

See Also
wthumb/3
wccreate/8
wcreate/8
wdcreate/7

write/1

write a term to the current output stream

write(*Term*)

?Term <term>

Writes the term *Term* to the current output stream. *Term* is output using the current operator declarations.

write does not output a dot ('.') after the term. If the term is to be read back in again, you must explicitly output the dot and following space after writing the term. To output this terminator use either of the following:

```
write('. ').
write('.'),nl.
```

Atoms that must be quoted on input (e.g. 'Paul', 'hello world') are not quoted when output using *write*. (If you want to guarantee being able to read an atom back again, you should output it with the *writeq* predicate.)

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
prompt/2
read/1
skip_term/0
sysops/0
vars/2
write_canonical/1
writeq/1

write_canonical/1

write a term to the current output stream in canonical form

write_canonical(*Term*)

+Term	<term>

Writes *Term* to the current output stream and quotes atoms and functors to make them acceptable as input to *read*/1.

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
prompt/2
read/1
skip_term/0
sysops/0
vars/2
write/1
writeq/1

writeq/1

write a quoted term to the current output stream

writeq(*Term*)

+Term	<atom> or <compound_term> or <list>

Writes *Term* to the current output stream. This predicate is the same as *write*/1, except that single quotes are put around any atoms in *Term* that would have to be quoted on input.

writeq/1 does not terminate the term with a full stop, so if you are writing to a file, and want to read the term back again you must follow the *writeq*/1 with either of the following:

```
write('. ').
write('.'),nl.
```

See Also

current_op/3
display/1
elex/1
eprint/1
eprint/2
eprint/3
eread/1
eread/2
etoks/1
etoks/2
ewrite/1
ewrite/2
ewrite/3
op/3
portray_clause/1
print/1
printq/1
prompt/2
read/1
skip_term/0
sysops/0
vars/2
write/1
write_canonical/1

wshow/2

get or set show or hide status

wshow(*Window*, *Status*)

+Window	<window_handle>
+Status	<integer> in the domain {0,1,2,3}

Get or set the status of the given window, enabling it to be hidden, minimised (iconised), normalised or maximised. The *Status* argument is a window visibility status value. This predicate causes a direct side effect on the window, whose status may be changed as specified.

If a window is initially hidden and then shown using *wshow/2*, it will gain focus *unless* it is disabled.

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wcreate/8
wdcreate/7
wdict/1
wenable/2
wfind/3
wfocus/1
wlink/3
wndhdl/2
wsize/5
wstyle/2
wtcreate/6
wtext/2
wucreate/6

wsize/5

get or set window size and position

wsize(*Window*, *Left*, *Top*, *Width*, *Height*)

+Window	<window_handle>
?Left	variable or <integer>
?Top	variable or <integer>
?Width	variable or <integer>
?Height	variable or <integer>

Get or set the *Left* and *Top* positions, and *Width* and *Height* dimensions, of the given *Window*. The numerical values in the Windows system are in pixel units and the DOS system in character units, where (0,0) is the left top of the screen. In general windows can be resized at any time, but in the Windows system it is recommended only to do this when they are in the normalised state (see *wshow/2*), because doing so when a window is maximised or minimised can confuse Windows' display driver.

Note: the position and size parameters are given in the standard Windows ordering, ie (Left, Top, Width, Height) and not in the older (Row, Column, Depth, Width) format used in other Prolog systems.

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wcreate/8
wdcreate/7
wdict/1
wenable/2
wfind/3
wfocus/1
wlink/3
wndhdl/2
wshow/2
wstyle/2
wtcreate/6
wtext/2
wucreate/6

wstyle/2

get or set window style

wstyle(*Window*, *Style*)

+Window	<window_handle>
+Style	<integer>

Get or set the *Style* of the given *Window*. Window styles should be changed with extreme caution, and should not be used to force alien styles onto windows. For example, changing the appearance of buttons is fine, but giving a button scroll bars is not.

One use of *wstyle/2* can be to create a window based on a given window's style. You should ensure that the windows are of the same class.

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wcreate/8
wdcreate/7
wdict/1
wenable/2
wfind/3
wfocus/1
wlink/3
wndhdl/2
wshow/2
wsize/5
wtcreate/6
wtext/2
wucreate/6

wtcreate/6

create a text window

wtcreate(*Name*, *Title*, *Left, Top, Width, Height*)

+Name	<window_handle>
+Title	<string>
+Left	<integer>
+Top	<integer>
+Width	<integer>
+Height	<integer>

Create a "text" window with the given *Name*, *Title*, *Left* - *Top* corner coordinates, *Width* and *Height* dimensions. *Name* should be an atom which is used from then on to refer to the window. Text windows contain an "editor" field that is automatically resized according to the resizing of the window.

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wcreate/8
wdcreate/7
wdict/1
wenable/2
wfind/3
wfocus/1
wlink/3
wndhdl/2
wshow/2
wsize/5
wstyle/2
wtext/2
wucreate/6

wtext/2

get or set the window text

wtext(*Window*, *Text*)

+Window	<window_handle>
+Text	<string>

Replace the text of the given *Window* to the given *Text*, or get the current *Text*. For top level and MDI child windows, the text is the window title (style permitting); for "button" and "static" control windows it is the window label, and for "edit" or "editor" control windows and the "edit" control components of "combobox" windows, it is the entire window contents. Note that, unlike *wedttxt/2*, this predicate works with all types of window, but instead of replacing the current selection it replaces the entire text.

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wcreate/8
wdcreate/7
wdict/1
wenable/2
wfind/3
wfocus/1
wlink/3
wndhdl/2
wshow/2
wsize/5
wstyle/2
wtcreate/6
wucreate/6

wthumb/3

get or set position of a scroll bar

wthumb(*Window*, *Type*, *Position*)

+Window	<window_handle>
+Type	<string>
?Position	<integer> or <variable>

Get or set the selection position of the elevator thumb in the given type of scroll bar in the given window. The *Position* argument is a signed 16-bit integer, limited by the scroll bar's range (see *wrange/4*). The *Type* argument is a scrollbar type value that shows whether a scroll bar is connected to a window (i.e. given as part of the style definition of that window) or stand-alone (a separate control item of type "scrollbar"). Windows addresses these two types of scrollbar differently, so *wthumb/3* needs to be provided with the appropriate type value.

See Also
wccreate/8
wcreate/8
wdcreate/7
wrange/4

wucreate/6

create a user MDI window

wucreate(*Name, Title, Left, Top, Width, Height*)

+Name	<window_handle>
+Title	<string>
+Left	<integer>
+Top	<integer>
+Width	<integer>
+Height	<integer>

Create a "user" MDI window with the given *Name*, *Title*, *Left* - *Top* corner coordinates, *Width* and *Height* dimensions. *Name* should be an atom which is used from then on to refer to the window. User windows are created with a system menu, a hide button, a maximize button and are re-sizeable. They do not contain any other controls. Note: if you put any control items in a user MDI window you must write your own code to handle the re-sizing of the window.

See Also

warea/5
wclass/2
wclose/1
wccreate/8
wcreate/8
wdcreate/7
wdict/1
wenable/2
wfind/3
wfocus/1
wlink/3
wndhdl/2
wshow/2
wsize/5
wstyle/2
wtcreate/6
wtext/2

wxclose/1

close a metafile

wxclose(*Metafile*)

+Metafile <atom>

Close the named memory metafile (*Metafile*), returning its resources to Windows, where *Metafile* was previously opened using *wxcreate/6* or *wxload/2*.

See Also
wxcreate/6
wxdict/1
wxload/2
wxsave/2

wxcreate/6

create a metafile

wxcreate(*Metafile, Glist, Left, Top, Right, Bottom*)

+Metafile	<atom>
+Glist	<list_of <compound_term> >
+Left	<integer>
+Top	<integer>
+Right	<integer>
+Bottom	<integer>

Create the named memory metafile (*Metafile*), using the given list of graphics commands (*Glist*), and *Left*, *Top*, *Right* and *Bottom* maximum extents. See *wgfx/2* for the list of supported functions.

See Also
wxclose/1
wxdict/1
wxload/2
wxsave/2

wxdict/1

return a dictionary of metafiles

wxdict(*XDict*)

-XDict <variable>

Return the dictionary of all currently defined memory metafiles as a list (*XDict*).

See Also
wxclose/1
wxcreate/6
wxload/2
wxsave/2

wxload/2

load a metafile from disk

wxload(*Metafile, DiskMetafile*)

+Metafile	<atom>
+DiskMetafile	<atom>

Load the named memory metafile (*Metafile*) from the specified disk metafile (*DiskMetafile*).

See Also
wxclose/1
wxcreate/6
wxdict/1
wxsave/2

wxsave/2

save a metafile to disk

wxsave(*Metafile, DiskMetafile*)

+Metafile	<atom>
+DiskMetafile	<atom>

Save the named memory metafile (*Metafile*) to the specified disk metafile (*DiskMetafile*).

See Also
wxclose/1
wxcreate/6
wxdict/1
wxload/2

Technical Support

The LPA Technical Support team is normally available during office hours (9am-5pm UK time), for phone, fax and email support.

When phoning, ask for LPA Prolog for Windows Technical Support. If no-one is available at that time, please leave your name and number and someone from Technical Support will get back to you. By far the best way to contact Technical Support is via email as this enables us to receive examples for testing and to send possible fixes.

If you need to contact the LPA Technical Support Team you can do so using the following numbers and addresses:

UK Phone and Fax

Phone: 0181 871 2016
Fax: 0181 874 0449

International Phone and Fax

Phone: +44 181 871 2016
Fax: +44 181 874 0449

email: lpa@cix.compulink.co.uk

World Wide Web Site

You are always welcome to visit our world wide web site at:

http://www.lpa.co.uk

Reporting Bugs and Problems

The following information will enable you to get the fastest and most efficient service from the LPA Technical Support team. There is a check list for the information that we normally require when dealing with problems and an example of a clearly presented report.

Screen dumps can provide useful information, as can DRWATSON log files, after serious Windows errors. Where possible, details of the host computer hardware and software can also help in bug analysis: the output of the MSD program (supplied as part of Windows) is very valuable in this respect. Ideally, a simple operating procedure, program or query should be presented which reliably reproduces the problem.

Please try to provide the following information:

1) **The exact product name (eg, LPA-PROLOG for Windows)**
2) **The serial number (eg, 0001234567)**
3) **The full version number (eg, 3.000)**
4) **The system compilation date (eg, 12 May 1995)**
5) **Settings of memory areas (eg, Bk=64, Lc=64, etc.)**
6) **The exact error message if one was given (eg, Error 30 File Handling Error)**
7) **If a GPF occurs please note down the exact address reported by Windows (eg. module <unknown> @ AB27E:3247)**

The first five pieces of information can be given by copying the Prolog banner, displayed in the console at startup, into the bug report. This banner can also be generated using the predicate ver/1.
Additionally, if a GPF has occurred please restart Windows, after noting down the exact address of the

GPF, and run the LPA Prolog for Windows application again. This time include the following command-line switch:

/V2

If the GPF re-occurs setting this switch provides some additional debugging information about the status of LPA Prolog for Windows. Note down the information that gets produced and send this to us along with some details regarding the circumstances of the GPF. Such as what type of process was happening when the GPF occurred, what other applications were running and any other information that you feel might be relevant.

Bug Reports

Bug reports should be clear, concise, and unambiguous, so as to maximise the chance of the problem being reproducible on LPA's computers. The following is an example of a good report.

=== START OF GOOD REPORT ===

Prolog Banner

LPA WIN-PROLOG 3.200 - S/N 0008429273 - 05 Jan 1996
Copyright (c) 1996 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=418 P=1502 S=63 I=64 O=64 Kb

Computer: See attached MSD report

Description of problem

I try to run Prolog from the DOS command line, with the command:

C> WIN PRO386W

Windows starts up as normal, followed by the LPA welcome banner, but then nothing further happens. If I press a key, the system just bleeps.

If I try to run Prolog from the Program Manager, by clicking on the LPA icon, everything works perfectly.

=== END OF GOOD REPORT ===